O'REILLY®


图灵程序设计丛书



Java攻略

Java常见问题的简单解法

Modern Java Recipes

70余个提炼自开发人员日常工作的范例，涵盖Java 8和Java 9新特性

[美] 肯·寇森 著

蒋楠 译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

蒋楠

毕业于电子科技大学、维多利亚大学，多年来致力于Web开发与软件架构设计，对算法、数据密集型应用、分布式数据库系统兴趣浓厚。非资深程序员，严肃马拉松跑者。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



图灵程序设计丛书

Java攻略

Java常见问题的简单解法

Modern Java Recipes:

Simple Solutions to Difficult Problems in Java 8 and 9

[美] 肯·寇森　著

蒋楠　译

Beijing • Boston • Farnham • Sebastopol • Tokyo　O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（C I P）数据

Java攻略 ： Java常见问题的简单解法 / (美) 肯・寇森 (Ken Kousen) 著 ; 蒋楠译. -- 北京 : 人民邮电出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48880-0

Ⅰ. ①J… Ⅱ. ①肯… ②蒋… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第155420号

内 容 提 要

本书旨在让读者迅速掌握 Java 8 和 Java 9 相关特性，并给出了 70 余个可以用于实际开发的示例，介绍了如何利用这些新特性解决这些问题，从而以更自然的方式让开发人员掌握 Java。

本书适合 Java 开发人员阅读。

◆ 著　　　[美] 肯・寇森
　译　　　蒋　楠
　责任编辑　朱　巍
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800 × 1000　1/16
　印张：15.5
　字数：366千字　　　2018年 8 月第 1 版
　印数：1 – 3 000册　　　2018年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-3407号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

Xanden，这本书是送你的礼物，想不到吧！

译者序

与其他语言相比，Java 最大的优势或许在于完善的生态系统：开发者所需要的一切，几乎都能从这个生态系统中找到。世界上累计有 150 亿台设备运行 Java，全球 Java 开发者的数量超过 1000 万人，Java 不仅构成了大量开源平台的基础，也已成为软件文化中不可或缺的一部分。

然而，作为一门诞生于 1995 年的语言，运行环境臃肿、代码库庞大等问题逐渐成为制约 Java 发展的瓶颈。对于稳定性与兼容性的顾虑，使得这门语言越来越难以大刀阔斧地改革。相当一部分业务服务仍然采用 Java 8 之前的版本构建，复杂的系统升级往往令维护人员望而却步。并非企业不想求变，而是求变的代价在某些时候显得异常高昂。

但 Oracle 从未停止探索的脚步，Java 9 的发布或许可以视作 Java 平台求变的开始。尽管社区对 Jigsaw 项目褒贬不一，模块化系统的意义仍然有待时间检验，不过 Oracle 意欲求变的决心由此可见一斑。而在 Java 9 面世之后，Oracle 加快了这门语言的迭代速度，版本发布周期改为半年一次，以便缩短开发者使用新功能的时间。Java 的发布速度经常受到诟病，这种改变或许有助于解决这个问题。

硬件厂商同样在探索前行。作为 JavaOne 2017 赞助商之一的英特尔在向量计算领域投入大量精力，发布了有助于 Java API 充分利用硬件向量计算性能的 Vector API。在推动整个生态系统发展方面，JavaOne 功不可没。

在软件工程师的职业生涯中，知识的“半衰期”通常为三年，这意味着我们掌握的一半知识在三年后将变得毫无价值。但这个行业本身就意味着不断充电与持续学习，技术大会或许是了解行业现状的有效途径。本书的诞生即源于 NFJS 巡回研讨会对作者的启发。NFJS 始于 2001 年，主要关注软件开发领域出现的最新技术，Java 与 JVM 是其中重点讨论的话题。

不过对 Java 开发者而言，掌握这门语言的各种技巧至关重要，夯实基础始终是首要任务。本书沿袭了 O’Reilly Media“编程食谱书”的一贯风格，将提炼自实际开发的问题以范例的形式展现给读者，使开发者对 Java 的关键知识点了然于心。O'Reilly Media 以出版技术类图书著称，其“动物书”系列与 Manning Publications 的“服饰书”系列备受开发人员的推崇。

非常感谢人民邮电出版社图灵文化发展有限公司的朱巍老师给予我翻译本书的机会，以及李冰、岳新欣、傅志红等各位编辑为本书所做的辛勤努力。虽然译者尽力而为，但水平有限，疏漏之处在所难免。恳请读者不吝赐教，提出宝贵的意见和建议。译者的联系方式：milesjiang314@gmail.com。

蒋楠

2018 年 6 月

序

毫无疑问，Java 8 引入的新特性（特别是 lambda 表达式和 Stream API）让这门语言经历了一次巨大的飞跃。多年来，我一直是 Java 8 的忠实用户，并在各种会议、研讨会以及博客上不遗余力地向开发人员介绍这些新特性。我很清楚，尽管 lambda 表达式和流让 Java 具备了更多函数式编程的特点（并行处理的威力也得以发挥），不过它们并非吸引开发人员的真正原因。新的习惯用法能让解决特定问题变得更为简单和高效，这才是 Java 8 备受追捧的根源所在。

作为一名程序员、演讲者和撰稿人，我不仅希望其他开发人员注意到 Java 语言的演变，还希望能展示这些演变是如何提高工作效率的。我们可以采用更简单的方法解决问题，甚至还能解决不同类型的问题。我之所以欣赏本书作者 Ken Kousen 的工作，是因为他在写作时严格遵循以下原则：帮助读者获取新知识，避免将时间花在已经了解或不需要的细节上。Ken 专注于对一线开发人员有价值的那些技术。

我第一次接触到 Ken 的工作，是他在 JavaOne 2013 会议上发表题为“Making Java Groovy: Simplify Your Java Development with Groovy”的演讲时。那时，我所在的团队正在为编写易读且有用的测试而殚精竭虑，我们所考虑的一种解决方案正是 Groovy。作为一名长期使用 Java 的程序员，我不愿意为了编写测试而去学习一门全新的语言，特别是我自认为已经了解如何编写测试时。然而，聆听 Ken 为 Java 程序员所做的 Groovy 介绍让我受益匪浅，他并未重复那些我已烂熟于心的内容，而是直入正题，使我迅速掌握了许多所需的知识。我意识到，选择合适的学习材料能极大地提高学习效率，我不必为了一个环节的应用而将一门语言的细枝末节全部吃透。因此，我立即购买了 Ken 撰写的 *Making Java Groovy* 一书[1]。

本书延续了类似的主题。作为经验丰富的开发者，我们无须像初学者一样学习 Java 8 和 Java 9 引入的所有新特性，也没有时间这样做。我们需要的是一本能迅速查找相关特性介绍的指南，并给出可以用于实际开发的示例。本书就是这样一本指南。书中范例来自开发人员在日常工作中遇到的问题，并介绍了如何利用 Java 8 和 Java 9 的新特性解决这些问

注 1：该书由 Manning Publications 于 2013 年 9 月出版，Ken 在 JavaOne 2013 会议上的演讲即以此为题。
——译者注

题，从而以更自然的方式让开发人员对这门语言的变化了然于心。我们可以举一反三，将所学的知识运用到实际开发中。

即便是 Java 8 和 Java 9 的长期使用者，依然可以从本书中受到启发。有关归约运算符的讨论切实加深了我对这种函数式编程风格的理解，而且我也无须重新理清思路。专门探讨 Java 9 新特性的章节正是开发人员所需要的，这些新特性尚未广为人知。本书提供了一种很好的方法，能够帮助读者快速有效地了解 Java 的最新发展。对所有希望提高自身知识水平的 Java 开发人员而言，本书堪称良师益友。

Trisha Gee

Java Champion

JetBrains 公司 Java 布道师

前言

与时俱进的Java

有时候，很难相信一门已保持了 20 年向后兼容性的语言会发生如此巨大的变化。在 Oracle 于 2014 年 3 月发布 Java SE 8 之前，作为最权威的服务器端编程语言，Java 已然赢得“21 世纪的 COBOL”这一美誉。Java 稳定且应用广泛，同时还不遗余力地追求性能。变化来得很慢，但还是来了。正因为如此，每当 Java 发布新版本时，企业的升级意愿并不迫切。

不过，在 Java SE 8 发布之后，一切都发生了改变。Java SE 8 将“Lambda 项目”（Project Lambda）纳入其中，这个重大的创新将函数式编程（functional programming）的概念引入这门杰出的面向对象语言。lambda 表达式、方法引用以及流从根本上改变了 Java 的习惯用法。自此之后，开发人员一直在努力跟上这门语言前进的步伐。

本书无意评判这些变化能否对 Java 开发有所促进，也无意探讨是否可以通过其他途径实现同样的目的。本书只是告诉读者，新特性已经存在，我们应该如何利用它们完成工作。这也是本书采用范例形式编写的原因。读者可以根据需要阅读本书，了解 Java 引入的新特性将如何帮助自己实现既定目标。

换言之，一旦掌握这种新的程序设计模型，就能享受它所带来的诸多优点。函数式代码往往更简单，而且更易于编写和理解。函数式编程强调不可变性（immutability），这使得编写的并发代码更简洁，调试和运行更容易成功。在 Java 初登舞台时，摩尔定律仍然有效：处理器的速度大约每 18 个月就提高一倍。而如今性能提升的根本在于，即使是手机也已大部分配备了多个处理器。

由于 Java 非常注重保持向后兼容性，不少企业和开发人员在迁移到 Java SE 8 时并未采用新的习惯用法。即便如此，Java SE 8 仍然是一个值得尝试的强大平台，而且 Oracle 已于 2015 年 4 月正式宣布停止对 Java 7 提供支持。

Java SE 8 发布至今已有几年时间，大部分 Java 开发人员目前都已转向 JDK 8。现在，深入了解 Java SE 8 对未来开发的意义和影响正当其时。希望本书能让这一过程变得更加容易。

目标读者

本书范例假定读者对 Java SE 8 之前的版本已有所了解。尽管不要求读者精通 Java，书中也会讨论某些较早的概念，但本书并非一本针对初学者的 Java 或面向对象编程教程。如果读者已使用 Java 开发过项目，并且熟悉标准库，阅读本书时应该不会感到困难。

本书涵盖与 Java SE 8 有关的几乎所有内容，并专门有一章介绍 Java 9 的新特性。如果希望了解 Java SE 8 新增的函数式习惯用法将如何改变代码的编写方式，这本包含丰富用例的教程是一个不错的选择。

Java 广泛应用于服务器端开发，拥有丰富的开源库和工具支持系统。Spring 和 Hibernate 是两种最流行的开源框架，二者只支持（或很快将只支持）Java 8 及以上的版本。如果读者计划使用 Java 8 或 Java 9 进行开发，本书讨论的范例或许能有所启发。

本书结构

本书以范例的形式编写和组织内容。但在讨论涉及 lambda 表达式、方法引用以及流的范例时，有时也会涉及其他内容。因此，前 6 章将介绍相关概念，不过读者无须以任何特定的顺序阅读。

各章主要内容如下。

- 第 1 章将介绍 lambda 表达式和方法引用的基础知识，然后讨论接口的新特性，包括**默认方法**和**静态方法**。此外，还将定义“函数式接口”，并解释它对于理解 lambda 表达式的重要性。
- 第 2 章主要介绍 Java 8 引入的 `java.util.function` 包，它包括 `Consumer`、`Supplier`、`Predicate` 以及 `Function` 这四类特殊的函数式接口，它们的应用贯穿于整个标准库。
- 第 3 章将介绍**流**的概念及其表示抽象的方法。流支持对数据进行转换和过滤，而非迭代地进行处理。这一章的范例将讨论与流相关的映射、过滤、归约等概念，它们与第 9 章介绍的并行和并发有密切的关系。
- 第 4 章主要介绍流数据的排序，并讨论如何将其转换为集合。这一章还将介绍分区和分组，它们将一般意义上的数据库操作转换为简单的库调用。
- 第 5 章是综合性的一章。在掌握 lambda 表达式、方法引用以及流的用法之后，读者将学习如何综合运用它们来解决某些有趣的问题。这一章还将讨论惰性、延迟执行、闭包复合等概念，以及异常处理这个令人头疼的问题。
- 第 6 章将讨论 Java 8 引入的颇具争议性的 `Optional` 类。这一章的范例将介绍 `Optional` 类的用法，以及如何创建实例并从中提取值。此外，我们将进一步讨论 `Optional` 类中 `map` 与 `flatMap` 操作所体现的函数式概念，以及它们与流中的 `map` 与 `flatMap` 操作有何不同。
- 第 7 章将介绍输入 / 输出流（与函数式流相对）的实际应用，以及 Java 8 针对文件和目录处理为标准库引入的一些函数式概念。
- 第 8 章将讨论 Java 8 引入的 Date-Time API，以及它如何取代传统且饱受诟病的 `Date` 类和 `Calendar` 类。这种新的 API 基于 Joda-Time 库，凝聚了大量开发人员多年的使用经验，已被重写为 `java.time` 包。坦率地讲，即便 Date-Time API 是 Java 8 新增的唯一特性，升级到 Java 8 也物有所值。

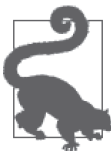
- 第 9 章主要介绍流模型的一种隐式承诺：通过一次方法调用，可以将串行流转换为并行流，从而充分利用计算机中所有可用的处理器。并发涉及的内容很多，这一章将重点介绍 Java 库的新增功能，这些功能便于用户进行试验，并评估成本和收益是否值得付出努力。
- 第 10 章将介绍 Java 9 引入的众多新特性，该版本于 2017 年 9 月 21 日正式发布。Jigsaw 本身的内容已可单独成书，但其基础概念十分清晰，这一章将对此进行介绍。其他范例将讨论接口中的私有方法，并介绍 `Stream`、`Collectors` 与 `Optional` 新增的各种方法，以及如何创建日期流[1]。
- 附录 A 将介绍 Java 中的泛型。泛型是 Java 1.5 引入的概念，但大部分开发人员对泛型只是略知皮毛，仅停留在完成工作所需的层面上。不过浏览 Java 8 和 Java 9 的 Javadoc 就会知道，这种日子已一去不复返。附录 A 旨在介绍如何阅读并解释 API，以帮助读者理解较为复杂的方法签名。

读者不必以任何特定的顺序阅读各章及其范例。各章之间互为补充，而且每个范例最后都包括指向其他范例的参考信息，所以从任何地方开始阅读均无不妥。章节分组是为了将相近的范例归类，但读者完全可以根据需要阅读所需的范例，以解决当前遇到的任何问题。

排版约定

本书使用以下排版约定。

- **黑体**
  表示新术语和重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键词等。
- 加粗等宽字体（**`constant width bold`**）
  表示命令以及其他需要用户输入的文字。
- 等宽斜体（*`constant width italic`*）
  表示这些值应该替换为用户输入，或根据上下文确定。



该图标表示提示或建议。

注 1：是的，我也希望能将讨论 Java 9 的章节排在第 9 章，不过仅仅为了偶然的对称性而对章节进行重新编排似乎不太合适。有这个脚注也就够了。



该图标表示一般性说明。



该图标表示警告。

示例代码

读者可以从 http://www.ituring.com.cn/book/2032 下载本书源代码，它们分别存储 Java 8 的相关范例（第 10 章除外）、Java 9 的相关范例以及范例 9.7 讨论的复杂示例，且均已配置为包含测试与构建文件的 Gradle 项目。

本书旨在帮助读者解决开发中遇到的问题。一般而言，读者不必获得 O'Reilly 的授权，就可以在自己的程序或文档中使用书中的示例代码。不过如果需要大量复制代码，则应该联系我们以获得许可。例如，读者可以直接在程序中使用本书的代码块。但是，销售或分发 O'Reilly 图书的配套光盘则需要获得许可。引用本书及其示例代码来解答问题，不需要获得许可。如果在自己的产品文档中大量使用书中的示例代码，则需要获得许可。

欢迎读者在使用本书的示例代码时注明出处，但这不是强制要求。通常要注明书名、作者、出版社和 ISBN。例如，"*Modern Java Recipes* by Ken Kousen (O'Reilly). Copyright 2017 Ken Kousen, 978-0-491-97317-2"。

如果读者认为对示例代码的使用不在合理使用和上述无须授权的范围之内，那么请通过 permissions@oreilly.com 联系我们。

O'Reilly Safari



Safari（之前称为 Safari Books Online）是一个为企业、政府、教育机构以及个人提供培训的会员制在线学习平台。

Safari 会员可以访问 250 多家出版商提供的数千种图书、培训视频、学习路径、互动教程以及精选列表等资源，这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

联系我们

如果读者对本书有任何评论或疑问，请通过以下地址联系我们。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

奥莱利技术咨询（北京）有限公司
北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

有关本书的技术性问题和建议，请通过邮箱 bookquestions@oreilly.com 联系我们。

欢迎访问 O'Reilly Media 网站，获取更多的图书、课程、会议信息以及最新动态。

我们的 Facebook 地址是 http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址是 http://www.youtube.com/oreillymedia。

致谢

2015 年 7 月下旬，我与 NFJS 巡回研讨会[2] 负责人 Jay Zimmerman 进行了一次谈话，这次谈话意外促成了本书的诞生。我当时是（目前也仍然是）NFJS 巡回研讨会的成员，那一年 Venkat Subramaniam 博士[3] 做了多次关于 Java 8 的演讲。由于 Subramaniam 博士决定在未来一年减少演讲的次数，Jay 询问我是否有兴趣从 2016 年初开始发表类似的演讲。我从 20 世纪 90 年代中期起就一直采用 Java 编写程序（所使用的第一个版本为 Java 1.0.6），对研究新 API 也抱有浓厚兴趣。念及此，我答应了 Jay 的请求。

几年来，我针对 Java 引入的函数式特性做了多次介绍。2016 年秋天，在完成上一本范例教程[4] 的写作后，我决定继续为 O'Reilly Media 撰写一本讨论 Java 的范例教程。那时我想当然地认为，这项工作应该是信手拈来。

注 2：NFJS 巡回研讨会（No Fluff Just Stuff Software Symposium Tour）于 2001 年在美国丹佛创办，重点关注现代软件开发与架构领域出现的最新技术和最佳实践，演讲者包括作者、咨询师、开发人员以及行业专家。从 2001 年至今，NFJS 巡回研讨会已经在全美各地举办了 500 多场活动，参与人数超过 8 万人。——译者注

注 3：Subramaniam 博士是 Agile Developer 公司创始人，休斯敦大学兼职教授。作为敏捷开发领域的权威人士，他培训了世界各地数以千计的软件开发人员。他还是一位多产的技术图书作者，所撰写的《Groovy 程序设计》一书是 Java 程序员学习 Groovy 的不二之选。另著有《高效程序员的 45 个习惯》《JavaScript 测试驱动开发》等书。——译者注

注 4：Gradle Recipes for Android，由 O'Reilly Media 出版，该书讨论了 Gradle 构建工具在 Android 项目中的应用。

著名科幻小说作家尼尔·盖曼（Neil Gaiman）曾表示，在完成《美国众神》一书的写作后，他认为自己已然了解如何写小说。不过他的朋友纠正说，他只是了解如何写**这部**小说而已。我现在理解了这句话的含义。本书最初计划写 150 页左右，包含 25 到 30 个范例，而成书接近 300 页，包含了 70 多个范例。不过这也使得本书的内容更为详尽和深入，比我预想的更有价值。

本书之所以能顺利出版，是因为我得到了不少人的帮助。Subramaniam 博士的演讲、著作以及与我的私下交流让我受益匪浅，他还欣然承担了本书的技术审阅工作。因此，若仍有疏漏，责任在他。（玩笑而已，责任当然应由我来承担，但请读者不要告诉 Subramaniam 博士我承认这一点。）

我在写作本书的过程中时常得到 Tim Yates 的帮助，我对此表示由衷的感谢。Tim 是我遇到的最好的程序员之一，他为 Groovy 社区所做的贡献远近闻名。不过 Tim 的才华不仅限于 Groovy，他是个多面手，这从他在 Stack Overflow 的评价可见一斑。我在 NFJS 巡回研讨会上发表 Java 8 的演讲时与 Rod Hilton 相识，他也为书稿提出了不少反馈意见。Tim 和 Rob 的建议都弥足珍贵。

我有幸与优秀的 O'Reilly Media 团队合作，出版了两本图书、十几部视频教程以及大量在线培训课程，读者可以通过在线平台 O'Reilly Safari 获取这些资源。Brian Foster 不仅提供了始终如一的支持，也展现了他克服官僚作风的出众能力。我在撰写上一本图书时与 Brian 相识，尽管他并未担任本书的编辑，但他的帮助使我受益良多，我们始终保持着良好的关系。

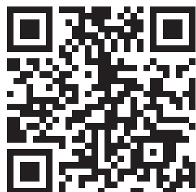
对于本书篇幅翻番，责任编辑 Jeff Bleiel 给予了充分理解，并对如何组织书稿提供了不少建议。我非常高兴与 Jeff 共事，也希望我们今后能继续合作。

我对以下 NFJS 演讲嘉宾表示感谢，他们不断的鼓励是我前行的动力：Nate Schutta、Michael Carducci、Matt Stine、Brian Sletten、Mark Richards、Pratik Patel、Neal Ford、Craig Walls、Raju Gandhi、Kirk Knoernschild、Dan“the Man”Hinojosa 以及 Janelle Klein。无论写书还是教学培训（我的实际工作），都是孤独的事业。我在社区结识了很多朋友和同事，大家共同讨论，一起玩耍。于我而言，这是一种有益的体验。

最后，我要对妻子 Ginger 和儿子 Xander 表示诚挚的谢意。没有你们的支持和体谅，就没有我今天的成就，这一点随着岁月的流逝变得越来越明显。任何言语都难以表达你们之于我的重要意义。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章

基础知识

Java 8 的最大变化是引入了函数式编程（functional programming）的概念。具体而言，Java 8 增加了 lambda 表达式（lambda expression）、方法引用（method reference）和流（stream）。

如果读者尚未接触过这些新增的函数式特性，或许会惊讶于写出的代码和之前大相径庭。Java 8 的变化堪称迄今为止这门语言最大的变化。从许多方面来说，这与学习一门全新的语言并无二致。

读者或许会问，这样做的目的是什么？为什么要对一门已有 20 年历史且计划保持向后兼容性的语言做出如此巨大的改变？一门为各方所公认的成熟语言，是否需要这些重大的修改？作为这些年来最成功的面向对象语言之一，Java 为何要转向函数式范式？

答案在于，软件开发领域已经发生变化，希望今后依然立于不败之地的语言同样需要适应这种变化。20 世纪 90 年代中期，在 Java 刚问世时，摩尔定律[1]仍然被奉为金科玉律。人们只需等上几年，计算机的运行速度就会提高一倍。

如今，硬件不再依赖于通过增加芯片密度来提高速度。相反，连手机都已大部分配备了多核处理器，这意味着编写的软件需要具备在多处理器环境下运行的能力。函数式编程强调“纯”函数（基于相同的输入将返回相同的结果，无副作用存在）以及不可变性，从而简化了并行环境下的编程。如果不引入任何共享、可变的状态，且程序可以被分解为若干简单函数的集合，就更容易理解并预测程序的行为。

不过，本书并非一本介绍 Haskell、Erlang、Frege 或任何其他函数式编程语言的教程，而是侧重于探讨 Java 以及它所引入的函数式概念。从本质上讲，Java 仍然是一门面向对象的语言。

注 1：由仙童半导体公司（Fairchild Semiconductor）和英特尔联合创始人戈登 • 摩尔（Gordon Moore）提出。根据观察，大约每 18 个月，封装到集成电路中的晶体管数量将增加一倍。参见维基百科的“摩尔定律”词条。

Java 目前支持 lambda 表达式，它本质上是被视为一类对象（first-class object）的方法。Java 还增加了方法引用，允许在任何需要 lambda 表达式的场合使用现有的方法。为利用 lambda 表达式和方法引用，Java 引入了流模型。流模型生成元素并通过流水线（pipeline）进行传递和过滤，无须修改原始源代码。

本章的范例将介绍 lambda 表达式、方法引用与函数式接口（functional interface）的基本语法，并讨论接口中的默认方法和静态方法。有关流的详细讨论，参见第 3 章。

1.1　lambda表达式

问题

用户希望在代码中使用 lambda 表达式。

方案

使用某种 lambda 表达式语法，并将结果赋给函数式接口类型的引用。

讨论

函数式接口是一种包含单一抽象方法（single abstract method）的接口。类通过为接口中的所有方法提供实现来实现任何接口，这可以通过顶级类（top-level class）、内部类甚至匿名内部类完成。

以 `Runnable` 接口为例，它从 Java 1.0 开始就已存在。该接口包含的单一抽象方法是 `run`，它不传入任何参数并返回 `void`。`Thread` 类构造函数传入 `Runnable` 作为参数，例 1-1 显示了 `Runnable` 接口的匿名内部类实现。

例 1-1　`Runnable` 接口的匿名内部类实现

```
public class RunnableDemo {
    public static void main(String[] args) {
        new Thread(new Runnable() {  ❶
            @Override
            public void run() {
                System.out.println(
                    "inside runnable using an anonymous inner class");
            }
        }).start();
    }
}
```

❶ 匿名内部类

匿名内部类语法以关键字 `new` 开头，后面跟着 `Runnable` 接口名以及英文小括号，表示定义一个实现该接口但没有显式名（explicit name）的类。大括号（`{}`）中的代码重写 `run` 方法，将字符串打印到控制台。

例 1-2 中的代码采用 lambda 表达式，对例 1-1 进行了改写。

例 1-2　在 Thread 构造函数中使用 lambda 表达式

```
new Thread(() -> System.out.println(
    "inside Thread constructor using lambda")).start();
```

上述代码使用箭头将参数与函数体隔开（由于没有参数，这里只使用一对空括号）。可以看到，函数体只包含一行代码，所以不需要大括号。这种语法被称为 lambda 表达式。注意，任何表达式求值都会自动返回。在本例中，由于 `println` 方法返回的是 `void`，所以该表达式同样会返回 `void`，与 `run` 方法的返回类型相匹配。

lambda 表达式必须匹配接口中单一抽象方法签名的参数类型和返回类型，这被称为与方法签名**兼容**。因此，lambda 表达式属于接口方法的实现，可以将其赋给该接口类型的引用。

例 1-3 显示了赋给某个变量的 lambda 表达式。

例 1-3　将 lambda 表达式赋给变量

```
Runnable r = () -> System.out.println(
    "lambda expression implementing the run method");
new Thread(r).start();
```



Java 库中不存在名为 `Lambda` 的类，lambda 表达式只能被赋给函数式接口引用。

“将 lambda 表达式赋给函数式接口”与“lambda 表达式属于函数式接口中单一抽象方法的实现”表示相同的含义。我们可以将 lambda 表达式视为实现接口的匿名内部类的主体。这就是 lambda 表达式必须与抽象方法兼容的原因，其参数类型和返回类型必须匹配该方法的签名。注意，所实现方法的名称并不重要，它不会作为 lambda 表达式语法的一部分出现在代码中。

因为 `run` 方法不传入参数，并且返回 `void`，所以本例特别简单。函数式接口 `java.io.FilenameFilter` 从 Java 1.0 开始就是 Java 标准库的一部分，该接口的实例被用作 `File.list` 方法的参数，只有满足该方法的文件才会被返回。

根据 Javadoc 的描述，`FilenameFilter` 接口包含单一抽象方法 `accept`，它的签名如下：

```
boolean accept(File dir, String name)
```

其中，`File` 参数用于指定文件所在的目录，`String` 用于指定文件名。

例 1-4 采用匿名内部类来实现 `FilenameFilter` 接口，只返回 Java 源文件。

例 1-4　FilenameFilter 的匿名内部类实现

```
File directory = new File("./src/main/java");

String[] names = directory.list(new FilenameFilter() {  ❶
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(".java");
    }
});
System.out.println(Arrays.asList(names));
```

❶ 匿名内部类

在例 1-4 中，如果文件名以 `.java` 结尾，`accept` 方法将返回 `true`，否则返回 `false`。

而例 1-5 采用 lambda 表达式实现 `FilenameFilter` 接口。

例 1-5　`FilenameFilter` 接口的 lambda 表达式实现

```
File directory = new File("./src/main/java");

String[] names = directory.list((dir, name) -> name.endsWith(".java"));  ❶
    System.out.println(Arrays.asList(names));
}
```

❶ lambda 表达式

可以看到，代码要简单得多。参数包含在小括号中，但并未声明类型。在编译时，编译器发现 `list` 方法传入一个 `FilenameFilter` 类型的参数，从而获知其单一抽象方法 `accept` 的签名，进而了解 `accept` 的参数为 `File` 和 `String`，因此兼容的 lambda 表达式参数必须匹配这些类型。由于 `accept` 方法的返回类型是布尔值，所以箭头右侧的表达式也必须返回布尔值。

如例 1-6 所示，我们也可以在代码中指定数据类型。

例 1-6　具有显式数据类型的 lambda 表达式

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) ->  ❶
    name.endsWith(".java"));
```

❶ 显式数据类型

此外，如果 lambda 表达式的实现多于一行，则需要使用大括号和显式返回语句，如例 1-7 所示。

例 1-7　lambda 代码块

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) -> {  ❶
    return name.endsWith(".java");
});
System.out.println(Arrays.asList(names));
```

❶ 代码块语法

这就是 lambda 代码块（block lambda）。在本例中，虽然代码主体只有一行，但可以使用大括号将多个语句括起来。注意，不能省略 `return` 关键字。

lambda 表达式在任何情况下都不能脱离**上下文**存在，上下文指定了将表达式赋给哪个函数式接口。lambda 表达式既可以是方法的参数，也可以是方法的返回类型，还可以被赋给引用。无论哪种情况，赋值类型必须为函数式接口。

1.2　方法引用

问题

用户希望使用方法引用来访问某个现有的方法，并将其作为 lambda 表达式进行处理。

方案

使用双冒号表示法（::）将实例引用或类名与方法分开。

讨论

如果说 lambda 表达式本质上是将方法作为对象进行处理，那么方法引用就是将现有方法作为 lambda 表达式进行处理。

例如，`java.lang.Iterable` 接口的 `forEach` 方法传入 `Consumer` 作为参数。如例 1-8 所示，`Consumer` 可以作为 lambda 表达式或方法引用来实现。

例 1-8　利用方法引用访问 `println` 方法

```
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(x -> System.out.println(x));     ❶

Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(System.out::println);            ❷

Consumer<Integer> printer = System.out::println;  ❸
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(printer);
```

❶ 使用 lambda 表达式

❷ 使用方法引用

❸ 将方法引用赋给函数式接口

双冒号表示法在 `System.out` 实例上提供了对 `println` 方法的引用，它属于 `PrintStream` 类型的引用。方法引用的末尾无须括号。在本例中，程序将流的所有元素打印到标准输出。[2]



如果编写一个只有一行的 lambda 表达式来调用方法，不妨考虑改用等价的方法引用。

与 lambda 语法相比，方法引用具有几个（不那么显著的）优点。首先，方法引用往往更短。其次，方法引用通常包括含有该方法的类的名称。这两点使得代码更易于阅读。

注 2：讨论 lambda 表达式或方法引用时，很难不涉及流，第 3 章将专门讨论流。目前可以这样理解：流依次产生一系列元素，但不会将它们存储在任何位置，也不会对原始源进行修改。

如例 1-9 所示，方法引用也可以和静态方法一起使用。

例 1-9　在静态方法中使用方法引用

```
Stream.generate(Math::random)      ❶
        .limit(10)
        .forEach(System.out::println); ❷
```

❶ 静态方法

❷ 实例方法

`Stream` 接口定义的 `generate` 方法传入 `Supplier` 作为参数。`Supplier` 是一个函数式接口，其单一抽象方法 `get` 不传入任何参数且只生成一个结果。`Math` 类的 `random` 方法与 `get` 方法的签名相互兼容，因为 `random` 方法同样不传入任何参数，且产生一个 0 到 1 之间、均匀分布的双精度伪随机数。方法引用 `Math::random` 表示该方法是 `Supplier` 接口的实现。

由于 `Stream.generate` 方法产生的是一个无限流（infinite stream），我们通过 `limit` 方法限定只生成 10 个值，然后使用方法引用 `System.out::println` 将这些值打印到标准输出，作为 `Consumer` 的实现。

语法

方法引用包括以下三种形式，其中一种存在一定的误导性。

`object::instanceMethod`
　引用特定对象的实例方法，如 `System.out::println`。

`Class::staticMethod`
　引用静态方法，如 `Math::max`。

`Class::instanceMethod`
　调用特定类型的任意对象的实例方法，如 `String::length`。

最后一种形式或许令人困惑，因为在 Java 开发中，一般只通过类名来调用静态方法。请记住，lambda 表达式和方法引用在任何情况下都不能脱离上下文存在。以对象引用为例，上下文提供了方法的参数。对于 `System.out::println`，等效的 lambda 表达式为（如例 1-8 中的上下文所示）：

```
// 相当于System.out::println
x -> System.out.println(x)
```

上下文提供了 `x` 的值，它被用作方法的参数。

静态方法 `max` 与之类似：

```
// 相当于Math::max
(x,y) -> Math.max(x,y)
```

此时，上下文需要提供两个参数，lambda 表达式返回较大的参数。

“通过类名来调用实例方法”语法的解释有所不同，其等效的 lambda 表达式为：

```
// 相当于String::length
x -> x.length()
```

此时，当上下文提供 x 的值时，它将用作方法的目标而非参数。



如果通过类名引用一个传入多个参数的方法，则上下文提供的第一个元素将作为方法的目标，其他元素作为方法的参数。

例 1-10 显示了相应的代码。

例 1-10　从类引用（class reference）调用多参数实例方法

```
List<String> strings =
    Arrays.asList("this", "is", "a", "list", "of", "strings");
List<String> sorted = strings.stream()
        .sorted((s1, s2) -> s1.compareTo(s2))  ❶
        .collect(Collectors.toList());

List<String> sorted = strings.stream()
        .sorted(String::compareTo)             ❶
        .collect(Collectors.toList());
```

❶ 方法引用及其等效的 lambda 表达式

`Stream` 接口定义的 `sorted` 方法传入 `Comparator<T>` 作为参数，其单一抽象方法为 `int compare(String other)`。`sorted` 方法将每对字符串提供给比较器，并根据返回整数的符号对它们进行排序。在本例中，上下文是一对字符串。方法引用语法（采用类名 `String`）调用第一个元素（lambda 表达式中的 `s1`）的 `compareTo` 方法，并使用第二个元素 `s2` 作为该方法的参数。

在流处理中，如果需要处理一系列输入，则会频繁使用方法引用中的类名来访问实例方法。例 1-11 显示了对流中各个 `String` 调用 `length` 方法。

例 1-11　使用方法引用在 `String` 上调用 `length` 方法

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(String::length)           ❶
        .forEach(System.out::println); ❷
```

❶ 通过类名访问实例方法

❷ 通过对象引用访问实例方法

程序调用 `length` 方法将每个字符串转换为一个整数，然后打印所有结果。

方法引用本质上属于 lambda 表达式的一种简化语法。lambda 表达式在实际中更常见，因为每个方法引用都存在一个等效的 lambda 表达式，反之则不然。对于例 1-11 中的方法引用，其等效的 lambda 表达式如例 1-12 所示。

例 1-12　方法引用的等效 lambda 表达式

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(s -> s.length())
        .forEach(x -> System.out.println(x));
```

对任何 lambda 表达式来说，上下文都很重要。为避免歧义，不妨在方法引用的左侧使用 `this` 或 `super`。

另见

用户也可以使用方法引用语法来调用构造函数，相关讨论参见范例 1.3。第 2 章将讨论 `java.util.function` 包以及本范例中出现的 `Supplier` 接口。

1.3　构造函数引用

问题

用户希望将方法引用作为流的流水线（stream pipeline）的一部分，以实例化某个对象。

方案

在方法引用中使用 `new` 关键字。

讨论

在讨论 Java 8 引入的新语法时，通常会提及 lambda 表达式、方法引用以及流。例如，我们希望将一份人员列表转换为相应的姓名列表。例 1-13 的代码段显示了解决这个问题的两种方案。

例 1-13　将人员列表转换为姓名列表

```
List<String> names = people.stream()
    .map(person -> person.getName()) ❶
    .collect(Collectors.toList());

// 或者采用以下方案

List<String> names = people.stream()
    .map(Person::getName)             ❷
    .collect(Collectors.toList());
```

❶ lambda 表达式

❷ 方法引用

那么，是否存在其他解决方案呢？如何根据字符串列表来创建相应的 `Person` 引用列表呢？尽管仍然可以使用方法引用，不过这次我们改用关键字 `new`，这种语法称为**构造函数引用**（constructor reference）。

为了说明构造函数引用的用法，我们首先创建一个 `Person` 类，它是最简单的 Java 对象（POJO）。`Person` 类的唯一作用是包装一个名为 `name` 的简单字符串特性，如例 1-14 所示。

例 1-14　Person 类

```java
public class Person {
    private String name;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    // getter和setter

    // equals、hashCode与toString方法
}
```

如例 1-15 所示，给定一个字符串集合，通过 lambda 表达式或构造函数引用，可以将其中的每个字符串映射到 `Person` 类。

例 1-15　将字符串转换为 Person 实例

```java
List<String> names =
    Arrays.asList("Grace Hopper", "Barbara Liskov", "Ada Lovelace",
        "Karen Spärck Jones");

List<Person> people = names.stream()
    .map(name -> new Person(name)) ❶
    .collect(Collectors.toList());

// 或采用以下方案

List<Person> people = names.stream()
    .map(Person::new)              ❷
    .collect(Collectors.toList());
```

❶ 使用 lambda 表达式来调用构造函数

❷ 使用构造函数引用来实例化 `Person`

`Person::new` 的作用是引用 `Person` 类中的构造函数。与所有 lambda 表达式类似，由上下文决定执行哪个构造函数。由于上下文提供了一个字符串，使用单参数的 `String` 构造函数。

1. 复制构造函数

复制构造函数（copy constructor）传入一个 `Person` 参数，并返回一个具有相同特性的新 `Person`，如例 1-16 所示。

例 1-16　Person 的复制构造函数

```java
public Person(Person p) {
    this.name = p.name;
}
```

如果需要将流代码从原始实例中分离出来，复制构造函数将很有用。假设我们有一个人员列表，先将其转换为流，再转换回列表，那么引用不会发生变化，如例 1-17 所示。

例 1-17　将列表转换为流，再转换回列表

```
Person before = new Person("Grace Hopper");

List<Person> people = Stream.of(before)
    .collect(Collectors.toList());
Person after = people.get(0);

assertTrue(before == after);                          ❶

before.setName("Grace Murray Hopper");                ❷
assertEquals("Grace Murray Hopper", after.getName()); ❸
```

❶ 对象相同

❷ 使用 `before` 引用修改人名

❸ `after` 引用中的人名已被修改

如例 1-18 所示，可以通过复制构造函数来切断连接。

例 1-18　使用复制构造函数

```
people = Stream.of(before)
      .map(Person::new)            ❶
      .collect(Collectors.toList());

after = people.get(0);
assertFalse(before == after);      ❷
assertEquals(before, after);       ❸

before.setName("Rear Admiral Dr. Grace Murray Hopper");
assertFalse(before.equals(after));
```

❶ 使用复制构造函数

❷ 对象不同

❸ 但二者是等效的

可以看到，当调用 `map` 方法时，上下文是 `Person` 实例的流。因此，`Person::new` 调用构造函数，它传入一个 `Person` 实例并返回一个等效的新实例，同时切断了 `before` 和 `after` 引用之间的连接。[3]

2. 可变参数构造函数

接下来，我们为 `Person` POJO 添加一个可变参数构造函数（varargs constructor），如例 1-19 所示。

注 3：需要说明的是，将葛丽丝 • 霍普（Grace Hopper）将军作为本书代码中的“对象”绝无冒犯之意。作者深信，尽管霍普将军已于 1992 年去世，她的水平仍然是作者所无法企及的。（葛丽丝 • 霍普是美国海军准将，也是全球最早的程序员之一。霍普开发了 COBOL 语言，被誉为“COBOL 之母”，计算机术语 bug 和 debug 也是由霍普团队首先使用并流传开来的。——译者注）

例 1-19　构造函数 Person 传入 String 的可变参数列表

```
public Person(String... names) {
    this.name = Arrays.stream(names)
                      .collect(Collectors.joining(" "));
}
```

上述构造函数传入零个或多个字符串参数，并使用空格作为定界符将这些参数拼接在一起。

那么，如何调用这个构造函数呢？任何传入零个或多个字符串参数（由逗号隔开）的客户端都会调用这个构造函数。一种方案是利用 String 类定义的 split 方法，它传入一个定界符并返回一个 String 数组。

```
String[] split(String delimiter)
```

因此，例 1-20 中的代码将列表中的每个字符串拆分为单个单词，并调用可变参数构造函数。

例 1-20　可变参数构造函数的应用

```
names.stream()                        ❶
    .map(name -> name.split(" "))     ❷
    .map(Person::new)                 ❸
    .collect(Collectors.toList());    ❹
```

❶ 创建字符串流

❷ 映射到字符串数组流

❸ 映射到 Person 流

❹ 收集到 Person 列表

在本例中，map 方法的上下文包含 Person::new 构造函数引用，它是一个字符串数组流，因此将调用可变参数构造函数。如果为该构造函数添加一个简单的打印语句：

```
System.out.println("Varargs ctor, names=" + Arrays.asList(names));
```

则输出如下结果：

```
Varargs ctor, names=[Grace, Hopper]
Varargs ctor, names=[Barbara, Liskov]
Varargs ctor, names=[Ada, Lovelace]
Varargs ctor, names=[Karen, Spärck, Jones]
```

3. 数组

构造函数引用也可以和数组一起使用。如果希望采用 Person 实例的数组（Person[]）而非列表，可以使用 Stream 接口定义的 toArray 方法，它的签名为：

```
<A> A[] toArray(IntFunction<A[]> generator)
```

toArray 方法采用 A 表示返回数组的泛型类型（generic type）。数组包含流的元素，由所提供的 generator 函数创建。我们甚至还能使用构造函数引用，如例 1-21 所示。

例 1-21　创建 Person 引用的数组

```
Person[] people = names.stream()
    .map(Person::new)         ❶
    .toArray(Person[]::new);  ❷
```

❶ Person 的构造函数引用

❷ Person 数组的构造函数引用

toArray 方法参数创建了一个大小合适的 Person 引用数组，并采用经过实例化的 Person 实例进行填充。

构造函数引用其实是方法引用的别称，通过关键字 new 调用构造函数。同样，由上下文决定调用哪个构造函数。在处理流时，构造函数引用可以提供很大的灵活性。

另见

有关方法引用的讨论，参见范例 1.2。

1.4　函数式接口

问题

用户希望使用现有的函数式接口，或编写自定义函数式接口。

方案

创建只包含单一抽象方法的接口，并为其添加 @FunctionalInterface 注解。

讨论

Java 8 引入的函数式接口是一种包含单一抽象方法的接口，因此可以作为 lambda 表达式或方法引用的目标。

关键字 abstract 在这里很重要。在 Java 8 之前，接口中的所有方法被默认视为抽象方法，不需要为它们添加 abstract。

我们定义一个名为 PalindromeChecker（回文检查器）的接口，如例 1-22 所示。

例 1-22　回文检查器接口

```
@FunctionalInterface
public interface PalindromeChecker {
    boolean isPalidrome(String s);
}
```

由于接口中的所有方法均为 public 方法[4]，可以省略访问修饰符，如同省略 abstract 关键字一样。

注 4：至少在 Java 9 之前，接口中也允许使用 private 方法。更多详细信息，参见范例 10.2。

由于 PalindromeChecker 仅包含一个抽象方法，它属于函数式接口。Java 8 在 java.lang 包中提供了 @FunctionalInterface 注解，可以应用到函数式接口，如例 1-22 所示。

@FunctionalInterface 注解并非必需，但使用它是一种好习惯，原因有两点。首先，@FunctionalInterface 注解会触发编译时校验（compile-time check），有助于确保接口符合要求。如果接口不包含或包含多个抽象方法，程序将提示编译错误。

其次，添加 @FunctionalInterface 注解后，会在 Javadoc 中生成以下语句：

```
Functional Interface:
This is a functional interface and can therefore be used as the assignment
target for a lambda expression or method reference.
```

函数式接口中同样可以使用 default 和 static 方法。由于这两种方法都有相应的实现，它们与“仅包含一个抽象方法”的要求并不矛盾。示例代码如例 1-23 所示。

例 1-23　MyInterface 是一个包含静态方法和默认方法的函数式接口

```
@FunctionalInterface
public interface MyInterface {
    int myMethod();           ❶
    // int myOtherMethod();   ❷

    default String sayHello() {
        return "Hello, World!";
    }

    static void myStaticMethod() {
        System.out.println("I'm a static method in an interface");
    }
}
```

❶ 单一抽象方法

❷ 如果这条语句未被注释掉，MyInterface 将不再是函数式接口

可以看到，如果 myOtherMethod 方法未被注释掉，MyInterface 就不再满足函数式接口的要求，@FunctionalInterface 注解将报错：“存在多个非重写的抽象方法。”

接口可以继承其他接口（甚至不止一个）。@FunctionalInterface 注解将对当前接口进行校验。因此，如果一个接口继承现有的函数式接口后，又添加了其他抽象方法，该接口就不再是函数式接口，如例 1-24 所示。

例 1-24　继承函数式接口的 MyChildInterface 不再属于函数式接口

```
public interface MyChildInterface extends MyInterface {
    int anotherMethod(); ❶
}
```

❶ 其他抽象方法

MyChildInterface 不属于函数式接口，因为它包含两个抽象方法：继承自 MyInterface 的 myMethod 和声明的 anotherMethod。即便没有添加 @FunctionalInterface 注解，代码也可以编译，因为 MyChildInterface 就是一个标准接口，但无法作为 lambda 表达式的目标。

此外，还有一种不太常见的情况值得注意。`Comparator` 接口用于排序，其他范例将对此进行讨论。查看 `Comparator` 接口的 Javadoc 信息，并点击 Abstract Methods（抽象方法）标签后，将显示以下信息（图 1-1）。

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
int	compare(T o1, T o2) Compares its two arguments for order.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.

图 1-1：`Comparator` 接口包含的抽象方法

可以看到，`Comparator` 接口包含两种抽象方法，且其中一种方法是在 `java.lang.Object` 类中实际实现的，那么 `Comparator` 为什么还属于函数式接口呢？

这里的特别之处在于，图中显示的 `equals` 方法来自 `Object` 类，因此已有一个默认的实现。根据 Javadoc 的描述，出于性能方面的考虑，用户可以提供满足相同契约（interface contract）的自定义 `equals` 方法，但不对该方法进行重写"始终是安全的"。

根据函数式接口的定义，`Object` 类中的方法与单一抽象方法的要求并不矛盾，因此 `Comparator` 仍然属于函数式接口。

另见

接口中默认方法的相关应用，参见范例 1.5。接口中静态方法的相关应用，参见范例 1.6。

1.5　接口中的默认方法

问题

用户希望在接口中提供方法的实现。

方案

将接口方法声明为 `default`，并以常规方式添加实现。

讨论

Java 之所以不支持多继承（multiple inheritance），是为了避免所谓的**钻石问题**（diamond problem）。考虑如图 1-2 所示的继承层次结构（有点类似 UML）。

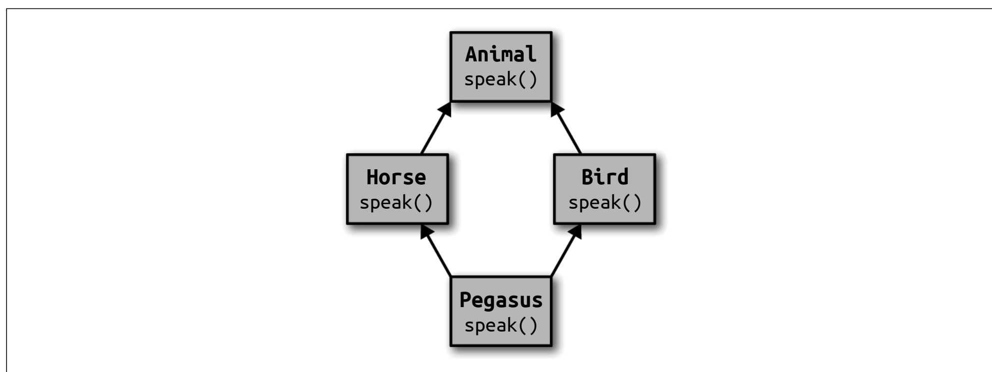


图 1-2：Animal 继承

Animal 类包括 Bird 和 Horse 两个子类，二者重写了 Animal 的 speak 方法：Horse 是“嘶嘶”（whinny），而 Bird 是“唧唧”（chirp）。那么 Pegasus（从 Horse 和 Bird 继承而来）[5] 呢？如果将 Animal 类型的引用赋给 Pegasus 的实例会怎样？ speak 方法又该返回什么呢？

```
Animal animal = new Pegaus();
animal.speak(); // 嘶嘶、唧唧还是其他声音？
```

不同语言处理这个问题的方法各不相同。例如，C++ 支持多继承，但如果某个类继承了相互冲突的实现则不会被编译。[6] 而在 Eiffel[7] 中，编译器允许用户选择所需的实现。

Java 禁止多继承。为避免一个类与多种类型都具有“某种”关系，Java 引入接口作为解决方案。由于接口只包含抽象方法，不会存在相互冲突的实现。接口之所以允许多继承，是因为只有方法签名被继承。

问题在于，如果永远无法在接口中实现方法，就会导致一些奇怪的情况出现。以 java.util.Collection 接口为例，它定义了以下方法：

```
boolean isEmpty()
int     size()
```

如果集合中没有元素，isEmpty 方法将返回 true，否则返回 false。而 size 方法返回集合中元素的数量。如例 1-25 所示，无论底层实现如何，都可以根据 size 立即实现 isEmpty 方法。

例 1-25　根据 size 实现 isEmpty 方法

```
public boolean isEmpty() {
    return size() == 0;
}
```

注 5：“一匹长有双翼的骏马。”（源自迪士尼电影《大力士海格力斯》，你不会没听说过希腊神话和海格力斯吧？）

注 6：但仍然可以使用虚继承（virtual inheritance）来解决这个问题。

注 7：Eiffel 或许对读者来说略显晦涩，它是面向对象编程的基础语言之一。感兴趣的话，可以参考 Bertrand Meyer 撰写的 *Object-Oriented Software Construction, Second Edition*，该书由 Prentice Hall 于 1997 年出版。

由于 Collection 是一个接口，不能对它进行这样的处理，但可以使用 Java 标准库提供的 java.util.AbstractCollection 类。它是一个抽象类，所包含的 isEmpty 方法与本例中 isEmpty 的实现完全相同。如果用户正在创建自定义的集合实现（collection implementation）且还没有超类，可以通过继承 AbstractCollection 类来获得 isEmpty 方法。不过如果已有超类，就必须改为实现 Collection 接口，且不要忘记提供自定义的 isEmpty 和 size 实现。

这些对经验丰富的 Java 开发人员而言很容易，但从 Java 8 开始，情况有所改变。目前只须将某个方法声明为 default 并提供一个实现，就能为接口方法添加实现。如例 1-26 所示，Employee 接口包含两种抽象方法和一种默认方法。

例 1-26　Employee 接口包含默认方法

```
public interface Employee {
    String getFirst();

    String getLast();

    void convertCaffeineToCodeForMoney();

    default String getName() {  ❶
        return String.format("%s %s", getFirst(), getLast());
    }
}
```

❶ 具有实现的默认方法

getName 方法由关键字 default 声明，其实现取决于 Employee 接口的另外两种抽象方法，即 getFirst 和 getLast。

为保持向后兼容性，Java 的许多现有接口都采用默认方法进行了增强。一般而言，为接口添加新方法会破坏所有现有的实现。如果添加的新方法被声明为默认方法，则所有现有的实现将继承新方法且仍然有效。这使得库维护者可以在 JDK 中添加新的默认方法，而不会破坏现有的实现。

例如，java.util.Collection 接口目前包含以下默认方法：

```
default boolean        removeIf(Predicate<? super E> filter)
default Stream<E>      stream()
default Stream<E>      parallelStream()
default Spliterator<E> spliterator()
```

removeIf 方法将删除集合中所有满足 Predicate[8] 参数的元素，如果删除了任何元素，该方法将返回 true。stream 和 parallelStream 方法用于创建流，二者属于工厂方法。spliterator 方法从实现 Spliterator 接口的类中返回一个对象，它对来自源的元素进行遍历和分区。

如例 1-27 所示，默认方法与其他方法的用法并无二致。

注 8：Predicate 是 java.util.function 包新增的一种函数式接口，相关讨论请参见范例 2.3。

例 1-27　默认方法的应用

```
List<Integer> nums = new ArrayList<>();
nums.add(-3);
nums.add(1);
nums.add(4);
nums.add(-1);
nums.add(5);
nums.add(9);
boolean removed = nums.removeIf(n -> n <= 0);  ❶
System.out.println("Elements were " + (removed ? "" : "NOT") + " removed");
nums.forEach(System.out::println);            ❷
```

❶ 使用 `Collection` 接口定义的默认方法 `removeIf`

❷ 使用 `Iterator` 接口定义的默认方法 `forEach`

如果一个类采用同一种默认方法实现了两个接口，会出现什么情况呢？范例 5.5 将讨论这个问题，不过简而言之，类可以实现方法本身。详细信息请参见范例 5.5。

另见

范例 5.5 将讨论一个类采用默认方法实现多个接口时需要遵守的规则。

1.6　接口中的静态方法

问题

用户希望为接口添加一个类级别（class level）的工具方法和相应的实现。

方案

将接口方法声明为 `static`，并以常规方式添加实现。

讨论

Java 类的静态成员是类级别的。换言之，静态成员与整个类相关联，而不是与特定的实例相关联。但从设计的角度看，静态成员在接口中的使用是有问题的，举例如下。

- 当多个不同的类实现接口时，类级别成员指的是什么？
- 类是否需要通过实现接口来使用静态方法？
- 类中的静态方法是通过类名访问的。如果类实现了一个接口，那么静态方法是通过类名还是接口名来调用呢？

为解决这些问题，Java 开发团队尝试了几种不同的方案。

在 Java 8 之前，接口完全不支持使用静态成员，不过这导致了**工具**类的产生，这是一种只包含静态方法的类。`java.util.Collections` 就是一种典型的工具类，它不仅包括用于排序和搜索的方法，也定义了采用同步或不可修改的类型包装集合的方法。在 `NIO` 包中，另一种工具

类是 `java.nio.file.Paths`，它只包括从字符串或 URI 中解析 `Path` 实例的两个静态方法。

而在 Java 8 中，我们可以随时为接口添加静态方法，步骤如下。

- 为方法添加 `static` 关键字。
- 提供一种无法被重写的实现。此时，静态方法类似于默认方法，包含在 Javadoc 的 Default Methods（默认标签）中。
- 通过接口名访问方法。类**不**需要通过实现接口来使用静态方法。

`java.util.Comparator` 接口定义的 `comparing` 方法就是一种实用的静态方法，它包括 `comparingInt`、`comparingLong`、`comparingDouble` 等三种基本变体。此外，`Comparator` 接口还包括 `naturalOrder` 和 `reverseOrder` 两种静态方法。例 1-28 给出了这些方法的用法。

例 1-28　字符串排序

```
List<String> bonds = Arrays.asList("Connery", "Lazenby", "Moore",
    "Dalton", "Brosnan", "Craig");

List<String> sorted = bonds.stream()
    .sorted(Comparator.naturalOrder())                ❶
    .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

sorted = bonds.stream()
    .sorted(Comparator.reverseOrder())                ❷
    .collect(Collectors.toList());
// [Moore, Lazenby, Dalton, Craig, Connery, Brosnan]

sorted = bonds.stream()
    .sorted(Comparator.comparing(String::toLowerCase)) ❸
    .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

sorted = bonds.stream()
    .sorted(Comparator.comparingInt(String::length))   ❹
    .collect(Collectors.toList());
// [Moore, Craig, Dalton, Connery, Lazenby, Brosnan]

sorted = bonds.stream()
    .sorted(Comparator.comparingInt(String::length)    ❺
        .thenComparing(Comparator.naturalOrder()))
    .collect(Collectors.toList());
// [Craig, Moore, Dalton, Brosnan, Connery, Lazenby]
```

❶ 自然顺序（字典序）

❷ 反向顺序（字典序）

❸ 按小写名称排序

❹ 按姓名长度排序

❺ 按姓名长度排序，如果长度相同则按字典序排序

本例显示了如何利用 `Comparator` 接口提供的静态方法对一份演员名单进行排序，名单中出现的演员是这些年来詹姆斯 • 邦德的扮演者。[9] 有关比较器的详细讨论，请参见范例 4.1。

由于接口中有静态方法，我们不必创建单独的工具类。但需要的话，仍然可以创建工具类。

请注意以下几点：

- 静态方法必须有一个实现
- 无法重写静态方法
- 通过接口名调用静态方法
- 无须实现接口以使用静态方法

另见

接口中静态方法的应用贯穿全书，有关 `Comparator` 接口定义的静态方法请参见范例 4.1。

注 9：我差点将伊德瑞斯 • 艾尔巴（Idris Elba）加入名单，但总算忍住没这么做。（艾尔巴是英国影星，因在《环太平洋》《雷神》等影片中饰演的角色为影迷所熟知，2014 年曾传出他将出演下一任邦德的消息——译者注。）

第 2 章

java.util.function包

第 1 章讨论了 lambda 表达式和方法引用的基本语法，二者在任何情况下都不能脱离上下文（context）而存在。lambda 表达式和方法引用总是被赋给函数式接口，它提供了所实现的单一抽象方法的信息。

Java 标准库中的许多接口仅包含一个抽象方法，它们属于函数式接口。 为此，Java 8 专门定义了 `java.util.function` 包，它仅包含可以在库的其余部分重用的函数式接口。

`java.util.function` 包中的接口分为四类，分别是 `Consumer`（消费型接口）、`Supplier`（供给型接口）、`Predicate`（谓词型接口）以及 `Function`（功能型接口）。`Consumer` 接口传入一个泛型参数（generic argument），不返回任何值；`Supplier` 接口不传入参数，返回一个值；`Predicate` 接口传入一个参数，返回一个布尔值；`Function` 接口传入一个参数，返回一个值。

每种基本接口还包含若干相关的接口。以 `Consumer` 接口为例，用于处理基本数据类型的是 `IntConsumer`、`LongConsumer` 和 `DoubleConsumer` 接口，`BiConsumer` 接口传入两个参数并返回 `void`。

虽然根据定义，这一章讨论的函数式接口只包含一个抽象方法，但大部分接口也包含声明为 `static` 或 `default` 的方法。对于开发人员而言，掌握这些方法有助于提高工作效率。

2.1 Consumer接口

问题

用户希望编写实现 `java.util.function.Consumer` 包的 lambda 表达式。

方案

使用 lambda 表达式或方法引用来实现 `void accept(T t)` 方法。

讨论

例 2-1 列出了 `Consumer` 接口定义的方法，其单一抽象方法为 `void accept(T t)`。

例 2-1　`Consumer` 接口定义的方法

```
        void        accept(T t)                         ❶
default Consumer<T> andThen(Consumer<? super T> after) ❷
```

❶ 单一抽象方法

❷ 用于复合操作的默认方法

`accept` 方法传入一个泛型参数并返回 `void`。在所有传入 `Consumer` 作为参数的方法中，最常见的是 `java.util.Iterable` 接口的默认 `forEach` 方法，如例 2-2 所示。

例 2-2　`Iterable` 接口定义的 `forEach` 方法

```
default void forEach(Consumer<? super T> action) ❶
```

❶ 将可迭代集合（iterable collection）中的所有元素传递给 `Consumer` 参数

如例 2-3 所示，所有线性集合通过对集合中每个元素执行给定的操作来实现 `Iterable` 接口。

例 2-3　打印集合中的元素

```
List<String> strings = Arrays.asList("this", "is", "a", "list", "of", "strings");

strings.forEach(new Consumer<String>() {      ❶
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
});

strings.forEach(s -> System.out.println(s));  ❷
strings.forEach(System.out::println);         ❸
```

❶ 匿名内部类实现

❷ lambda 表达式

❸ 方法引用

在本例中，由于 `accept` 方法只传入一个参数且不返回任何值，其签名与 lambda 表达式相符。通过 `System.out` 访问 `PrintStream` 类的 `println` 方法，它与 `Consumer` 相互兼容。因此，最后两条语句都可以作为 `Consumer` 参数的目标。

如表 2-1 所示，`java.util.function` 包还定义了三种 `Consumer<T>` 的基本变体，以及一种双参数形式。

表2-1：其他Consumer接口

接口	单一抽象方法
IntConsumer	void accept(int x)
DoubleConsumer	void accept(double x)
LongConsumer	void accept(long x)
BiConsumer	void accept(T t, U u)



Consumer 接口期望执行带有副作用的操作（即它可能会改变输入参数的内部状态），参见范例 2.3。

BiConsumer 接口的 accept 方法传入两个泛型参数，这两个泛型参数应为不同的类型。java.util.function 包定义了 BiConsumer 接口的三种变体，每种变体的第二个参数为基本数据类型。以 ObjIntConsumer 接口为例，其 accept 方法传入两个参数，分别为泛型参数和 int 参数。ObjLongConsumer 和 ObjDoubleConsumer 接口的定义与 ObjIntConsumer 类似。

标准库还支持 Consumer 接口的一些其他用法。

`Optional.ifPresent(Consumer<? super T> consumer)`
: 如果值存在，则调用指定的 consumer；否则不进行任何操作。

`Stream.forEach(Consumer<? super T> action)`
: 对流的每个元素执行操作[1]。Stream.forEachOrdered 方法与之类似,它根据元素的出现顺序（encounter order）访问元素。

`Stream.peek(Consumer<? super T> action)`
: 首先执行给定操作，再返回一个与现有流包含相同元素的流。peek 方法在调试中极为有用（参见范例 3.5）。

另见

Consumer 接口定义的 andThen 方法用于函数复合（function composition），详细讨论参见范例 5.8。有关 Stream.peek 方法的讨论参见范例 3.5。

2.2 Supplier接口

问题

用户希望实现 java.util.function.Supplier 接口。

注 1：这项操作十分常见，因此 Iterable 接口也直接定义了 forEach 方法。当源元素不是来自集合或需要创建并行流时，Stream.forEach 方法就很有用。

方案

使用 lambda 表达式或方法引用来实现 `T get()` 方法。

讨论

`Supplier` 接口相当简单，它不包含任何静态或默认方法，只有一个抽象方法 `T get()`。

为实现 `Supplier` 接口，需要提供一个不传入参数且返回泛型类型（generic type）的方法。根据 Javadoc 的描述，调用 `Supplier` 时，不要求每次都返回一个新的或不同的结果。

`Supplier` 的一种简单应用是 `Math.random` 方法，它不传入参数且返回 `double` 型数据。如例 2-4 所示，`Math.random` 方法可以被赋给 `Supplier` 引用并随时调用。

例 2-4　使用 `Math.random` 作为 `Supplier`

```
Logger logger = Logger.getLogger("...");

DoubleSupplier randomSupplier = new DoubleSupplier() { ❶
    @Override
    public double getAsDouble() {
        return Math.random();
    }
};

randomSupplier = () -> Math.random();                   ❷
randomSupplier = Math::random;                          ❸

logger.info(randomSupplier);
```

❶ 匿名内部类实现

❷ lambda 表达式

❸ 方法引用

`DoubleSupplier` 接口包含的单一抽象方法为 `getAsDouble`，它返回一个 `double` 型数据。表 2-2 列出了 `java.util.function` 包定义的其他相关 `Supplier` 接口。

表2-2：其他Supplier接口

接口	单一抽象方法
`IntSupplier`	`int getAsInt()`
`DoubleSupplier`	`double getAsDouble()`
`LongSupplier`	`long getAsLong()`
`BooleanSupplier`	`boolean getAsBoolean()`

`Supplier` 的一个主要用例是**延迟执行**（deferred execution）。`java.util.logging.Logger` 类定义的 `info` 方法传入 `Supplier`，仅当日志级别（log level）控制日志消息可见时，才调用其 `get` 方法（详见范例 5.7）。用户可以在自己的代码中应用延迟执行，以确保只在合适的情况下才从 `Supplier` 中检索值。

另一个用例是标准库中 `java.util.Optional` 类定义的 `orElseGet` 方法，它同样传入 `Supplier`。

有关 Optional 类的讨论请参见第 6 章，不过简而言之，Optional 类是一种容器对象（container object），要么包装值，要么为空。Optional 类通常用于在返回值可能合法为 null 时与用户进行通信，比如在空集中查找某个值。

我们以在一个集合中搜索名称为例，展示以上方法的应用，如例 2-5 所示。

例 2-5　在集合中查找名称

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "Inara",
    "Zoë", "Jayne", "Simon", "River", "Shepherd Book");

Optional<String> first = names.stream()
    .filter(name -> name.startsWith("C"))
    .findFirst();

System.out.println(first);                   ❶
System.out.println(first.orElse("None")); ❷

System.out.println(first.orElse(String.format("No result found in %s",
    names.stream().collect(Collectors.joining(", "))))); ❸

System.out.println(first.orElseGet(() ->
    String.format("No result found in %s",
    names.stream().collect(Collectors.joining(", "))))); ❹
```

❶ 打印 Optional.empty

❷ 打印字符串 "None"

❸ 即便找到指定的名称，仍然使用逗号分隔集合

❹ 仅当 Optional 为空时，才使用逗号分隔集合

Stream 接口的 findFirst 方法将返回有序流中出现的第一个元素。[2] 由于可以将流中的元素全部滤掉，findFirst 方法将返回一个 Optional。Optional 要么包含所需的元素，要么为空。在本例中，由于列表中的任何名称都不会传递给筛选器，所以结果是一个空 Optional。

Optional 类的 orElse 方法可以返回包含的元素，或者可以返回指定的默认值。虽然默认值为简单的字符串并无不妥，但如果需要经过处理才能返回值，则简单字符串的意义不大。

在本例中，返回值以逗号分隔的形式显示了名称的完整列表。无论 Optional 中是否包含值，orElse 方法都会创建完整的字符串。

而 orElseGet 方法传入 Supplier 作为参数。其优点在于，仅当 Optional 为空时，才会调用 Supplier 接口的 get 方法。换言之，除非确有必要，否则不会创建完整的名称字符串。

标准库还支持 Supplier 接口的其他一些用法。

- Optional 类的 orElseThrow 方法传入 Supplier<X extends Exception>，仅当发生异常时才会执行 Supplier。

注 2：流可能存在（也可能不存在）出现顺序，如同列表被假定为按索引排序而集合不是。这与元素的处理顺序可能有所不同。详见范例 3.9。

- 当 `Objects.requireNonNull(T obj, Supplier<String> messageSupplier)` 的第一个参数为空时，抛出自定义的 `NullPointerException`。
- `CompletableFuture.supplyAsync(Supplier<U> supplier)` 返回一个 `CompletableFuture`，它通过调用给定 `Supplier` 获得的值，以使得运行的任务异步完成。
- `Logger` 类的所有日志记录方法都有相应的重载形式，它传入 `Supplier<String>`，而不仅仅是一个字符串（详见范例 5.7）。

另见

有关日志记录方法的重载形式（传入 `Supplier`）请参见范例 5.7，有关查找集合中的第一个元素请参见范例 3.9，有关 `CompletableFuture` 类的讨论请参见第 9 章，有关 `Optional` 类的讨论请参见第 6 章。

2.3 Predicate接口

问题

用户希望使用 `java.util.function.Predicate` 接口筛选数据。

方案

使用 lambda 表达式或方法引用来实现 `boolean test(T t)` 方法。

讨论

`Predicate` 接口主要用于流的筛选。给定一个包含若干项的流，`Stream` 接口的 `filter` 方法传入 `Predicate` 并返回一个新的流，它仅包含满足给定谓词的项。

`Predicate` 接口包含的单一抽象方法为 `boolean test(T t)`，它传入一个泛型参数并返回 `true` 或 `false`。例 2-6 列出了 `Predicate` 接口定义的所有方法（包括静态和默认方法）。

例 2-6　Predicate 接口定义的方法

```
default    Predicate<T> and(Predicate<? super T> other)
static <T> Predicate<T> isEqual(Object targetRef)
default    Predicate<T> negate()
default    Predicate<T> or(Predicate<? super T> other)
boolean    test(T t)  ❶
```

❶ 单一抽象方法

给定一个名称集合，可以通过流处理找出所有具有特定长度的实例，如例 2-7 所示。

例 2-7　查找具有给定长度的字符串

```
public String getNamesOfLength(int length, String... names) {
    return Arrays.stream(names)
        .filter(s -> s.length() == length)   ❶
        .collect(Collectors.joining(", "));
}
```

❶ 满足给定长度字符串的谓词

或者，我们也可能只需要返回以特定字符串开头的名称，如例 2-8 所示。

例 2-8　查找以给定字符串开头的字符串

```
public String getNamesStartingWith(String s, String... names) {
    return Arrays.stream(names)
        .filter(str -> str.startsWith(s))   ❶
        .collect(Collectors.joining(", "));
}
```

❶ 返回以给定字符串开头的字符串

如果允许客户端指定条件，`Predicate` 的通用性会更强。例 2-9 显示了其中一种应用。

例 2-9　查找满足任意谓词的字符串

```
public class ImplementPredicate {
    public String getNamesSatisfyingCondition(
        Predicate<String> condition, String... names) {
            return Arrays.stream(names)
                .filter(condition)  ❶
                .collect(Collectors.joining(", "));
        }
    }

    // 其他方法
}
```

❶ 根据提供的谓词进行筛选

上述用法相当灵活，但依靠客户端自己编写所有谓词或许不太容易。一种方案是将常量添加到代表最常见情况的类中，如例 2-10 所示。

例 2-10　为常见情况添加常量

```
public class ImplementPredicate {
    public static final Predicate<String> LENGTH_FIVE = s -> s.length() == 5;
    public static final Predicate<String> STARTS_WITH_S =
        s -> s.startsWith("S");

    // 其余代码和之前一样
}
```

提供谓词作为参数的另一个优点是，可以使用默认方法 `and`、`or` 与 `negate`，并根据一系列单个元素来创建复合谓词（composite predicate）。

例 2-11 的测试用例展示了各种方法的应用。

例 2-11　针对谓词方法的 JUnit 测试

```
import static functionpackage.ImplementPredicate.*; ❶
import static org.junit.Assert.assertEquals;

// 其他导入

public class ImplementPredicateTest {
```

```
    private ImplementPredicate demo = new ImplementPredicate();
    private String[] names;

    @Before
    public void setUp() {
        names = Stream.of("Mal", "Wash", "Kaylee", "Inara", "Zoë",
            "Jayne", "Simon", "River", "Shepherd Book")
            .sorted()
            .toArray(String[]::new);
    }

    @Test
    public void getNamesOfLength5() throws Exception {
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesOfLength(5, names));
    }

    @Test
    public void getNamesStartingWithS() throws Exception {
        assertEquals("Shepherd Book, Simon",
            demo.getNamesStartingWith("S", names));
    }

    @Test
    public void getNamesSatisfyingCondition() throws Exception {
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesSatisfyingCondition(s -> s.length() == 5, names));
        assertEquals("Shepherd Book, Simon",
            demo.getNamesSatisfyingCondition(s -> s.startsWith("S"),
            names));
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesSatisfyingCondition(LENGTH_FIVE, names));
        assertEquals("Shepherd Book, Simon",
            demo.getNamesSatisfyingCondition(STARTS_WITH_S, names));
    }

    @Test
    public void composedPredicate() throws Exception {
        assertEquals("Simon",
            demo.getNamesSatisfyingCondition(
                LENGTH_FIVE.and(STARTS_WITH_S), names));   ❷
        assertEquals("Inara, Jayne, River, Shepherd Book, Simon",
            demo.getNamesSatisfyingCondition(
                LENGTH_FIVE.or(STARTS_WITH_S), names));    ❷
        assertEquals("Kaylee, Mal, Shepherd Book, Wash, Zoë",
            demo.getNamesSatisfyingCondition(LENGTH_FIVE.negate(), names)); ❸
    }
}
```

❶ 静态导入让使用常量更加简单

❷ 复合

❸ 否定

标准库还支持 Predicate 接口的其他一些用法。

`Optional.filter(Predicate<? super T> predicate)`
: 如果值存在且匹配某个给定的谓词，则返回描述该值的 Optional，否则返回一个空 Optional。

`Collection.removeIf(Predicate<? super E> filter)`
: 删除集合中所有满足谓词的元素。

`Stream.allMatch(Predicate<? super T> predicate)`
: 如果流的所有元素均满足给定的谓词，则返回 true。anyMatch 和 noneMatch 方法的用法与之类似。

`Collectors.partitioningBy(Predicate<? super T> predicate)`
: 返回一个 Collector，它将流分为两类（满足谓词和不满足谓词）。

只要流仅返回特定的元素，Predicate 接口就很有用。希望本范例能对读者有所启发，理解这种接口的用法。

另见

有关闭包复合（closure composition）的讨论请参见范例 5.8，有关 allMatch、anyMatch 与 noneMatch 方法的讨论请参见范例 3.10，有关分区和分组的讨论请参见范例 4.5。

2.4 Function接口

问题

用户希望实现 java.util.function.Function 接口，以便将输入参数转换为输出值。

方案

提供一个实现 R apply(T t) 方法的 lambda 表达式。

讨论

Function 接口包含的单一抽象方法为 apply，它可以将 T 类型的泛型输入参数转换为 R 类型的泛型输出值。例 2-12 列出了 Function 接口定义的所有方法。

例 2-12　Function 接口定义的方法

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
            R             apply(T t)
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
static  <T> Function<T,T> identity()
```

Function 最常见的用法是作为 Stream.map 方法的一个参数。例如，为了将 string 转换为整数，可以在每个实例上调用 length 方法，如例 2-13 所示。

例 2-13　将字符串映射到它们的长度

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "Inara",
        "Zoë", "Jayne", "Simon", "River", "Shepherd Book");

List<Integer> nameLengths = names.stream()
        .map(new Function<String, Integer>() { ❶
            @Override
            public Integer apply(String s) {
                return s.length();
            }
        })
        .collect(Collectors.toList());

nameLengths = names.stream()
        .map(s -> s.length())                ❷
        .collect(Collectors.toList());

nameLengths = names.stream()
        .map(String::length)                 ❸
        .collect(Collectors.toList());

System.out.printf("nameLengths = %s%n", nameLengths);
// nameLengths == [3, 4, 6, 5, 3, 5, 5, 5, 13]
```

❶ 匿名内部类

❷ lambda 表达式

❸ 方法引用

表 2-3 列出了输入和输出泛型类型的所有基本变体。

表2-3：其他Function接口

接口	单一抽象方法
`IntFunction`	`R apply(int value)`
`DoubleFunction`	`R apply(double value)`
`LongFunction`	`R apply(long value)`
`ToIntFunction`	`int applyAsInt(T value)`
`ToDoubleFunction`	`double applyAsDouble(T value)`
`ToLongFunction`	`long applyAsLong(T value)`
`DoubleToIntFunction`	`int applyAsInt(double value)`
`DoubleToLongFunction`	`long applyAsLong(double value)`
`IntToDoubleFunction`	`double applyAsDouble(int value)`
`IntToLongFunction`	`long applyAsLong(int value)`
`LongToDoubleFunction`	`double applyAsDouble(long value)`
`LongToIntFunction`	`int applyAsInt(long value)`
`BiFunction`	`R apply(T t, U u)`

在例 2-13 中，由于 `map` 方法返回的是基本数据类型 `int`，该方法的参数可能是 `ToIntFunction`。`Stream.mapToInt` 方法传入 `ToIntFunction` 作为参数，`mapToDouble` 和 `mapToLong` 方法与之类似。

mapToInt、mapToDouble 与 mapToLong 的返回类型分别为 IntStream、DoubleStream 和 LongStream。

那么，如果输入参数和返回类型相同呢？ java.util.function 包为此定义了 UnaryOperator 接口，以及相应的 IntUnaryOperator、DoubleUnaryOperator 和 LongUnaryOperator 接口，三者的输入和输出参数分别为 int、double 和 long。UnaryOperator 的一种应用是 StringBuilder 的 reverse 方法，因为输入类型和输出类型均为字符串。

BiFunction 接口定义了两个泛型输入类型和一个泛型输出类型，三者应为不同的类型。如果三者相同，可以使用 java.util.function 包定义的 BinaryOperator 接口。Math.max 就是一种二元运算符，其输入和输出为 int、double、float 或 long 型数据。相应地，java.util.function 包也定义了 IntBinaryOperator、DoubleBinaryOperator 与 LongBinaryOperator 接口。[3]

表 2-4 列出了 BiFunction 接口的所有基本变体。

表2-4：其他BiFunction接口

接口	单一抽象方法
ToIntBiFunction	int applyAsInt(T t, U u)
ToDoubleBiFunction	double applyAsDouble(T t, U u)
ToLongBiFunction	long applyAsLong(T t, U u)

尽管 Function 主要用于各种 Stream.map 方法，但这些方法也可能出现在其他上下文中，举例如下。

Map.computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
: 如果指定的键没有值，使用所提供的 Function 计算一个值并将其添加到 Map。

Comparator.comparing(Function<? super T,? extends U> keyExtractor)
: comparing 方法生成一个 Comparator，使用给定 Function 生成的键对集合进行排序。相关讨论请参见范例 4.1。

Comparator.thenComparing(Function<? super T,? extends U> keyExtractor)
: thenComparing 是一种实例方法，也可以用于排序。如果集合的首次排序返回相同的值，则使用另一种机制进行排序。

此外，Function 还广泛用于分组和下游收集器（downstream collector）的 Collectors 工具类中。

有关 andThen 和 compose 方法的讨论请参见范例 5.8。identity 方法实际上是一种简单的 lambda 表达式（e -> e），范例 4.3 将介绍其中一种应用。

另见

有关 Function.andThen 和 Function.compose 方法的应用请参见范例 5.8，有关 Function.identity 方法的应用请参见范例 4.3，有关下游收集器的讨论请参见范例 4.6，有关 computeIfAbsent 方法的讨论请参见范例 5.4，有关二元运算符的讨论请参见范例 3.3。

注 3：有关 Java 标准库中 BinaryOperator 用法的详细讨论，请参见范例 3.3。

第3章

流式操作

为支持函数式编程，Java 8 引入了新的流式隐喻（streaming metaphor）。流是一种元素序列，它不存储元素，也不会修改原始源。Java 的函数式编程通常涉及从某些数据源生成流，通过一系列称为**流水线**（pipeline）的中间操作（intermediate operation）传递元素，并利用**终止表达式**（terminal expression）完成这一过程。

流仅能使用一次。换言之，流在经过零个或多个中间操作并达到终止操作（terminal operation）后就会结束。如果希望再次对值进行处理，需要创建一个新的流。

此外，流是惰性的（lazy）。也就是说，流在达到终止条件（terminal condition）后才处理数据。范例 3.13 将讨论惰性流在实际中的应用。

本章的范例将介绍各种典型的流操作。

3.1　流的创建

问题

用户希望从数据源创建流。

方案

使用 `java.util.stream.Stream` 接口定义的各种静态工厂方法，以及 `java.lang.Iterable` 接口或 `java.util.Arrays` 类定义的 `stream` 方法。

讨论

Java 8 引入的 `Stream` 接口定义了多种用于创建流的静态方法。具体而言，可以采用 `Stream.of`、`Stream.iterate`、`Stream.generate` 等静态方法创建流。

`Stream.of` 方法传入元素的可变参数列表：

```
static <T> Stream<T> of(T... values)
```

在 Java 标准库中，`of` 方法的实现实际上被委托给 `java.util.Arrays` 类定义的 `stream` 方法，如例 3-1 所示。

例 3-1　`Stream.of` 方法的引用实现

```
@SafeVarargs
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```



`@SafeVarargs` 注解属于 Java 泛型的一部分，它在使用数组作为参数时出现。因为用户有可能将一个类型化数组（typed array）赋给一个 `Object` 数组，导致添加的元素引发类型安全问题。换言之，`@SafeVarargs` 注解构成了开发人员对类型安全的承诺。详见附录 A。

`Stream.of` 方法的简单应用如例 3-2 所示。



由于流在遇到终止表达式之前不会处理任何数据，本范例中的所有示例都会在末尾添加一个终止方法，如 `collect` 或 `forEach`。

例 3-2　利用 `Stream.of` 方法创建流

```
String names = Stream.of("Gomez", "Morticia", "Wednesday", "Pugsley")
    .collect(Collectors.joining(","));
System.out.println(names);
// 打印Gomez,Morticia,Wednesday,Pugsley
```

Java API 还定义了 `of` 方法的重载形式，它传入单个元素 `T t`，返回只包含一个元素的单例顺序流（singleton sequential stream）。

`Arrays.stream` 方法的应用如例 3-3 所示。

例 3-3　利用 `Arrays.stream` 方法创建流

```
String[] munsters = { "Herman", "Lily", "Eddie", "Marilyn", "Grandpa" };
names = Arrays.stream(munsters)
    .collect(Collectors.joining(","));
System.out.println(names);
// 打印Herman,Lily,Eddie,Marilyn,Grandpa
```

由于需要提前创建数组，上述方案略有不便，但足以满足可变参数列表的需要。Java API 定义了 `Arrays.stream` 方法的多种重载形式，用于处理 `int`、`long` 和 `double` 型数组，还定

义了本例使用的泛型类型。

Stream 接口定义的另一种静态工厂方法是 iterate，其签名如下：

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

根据 Javadoc 的描述，iterate 方法"返回一个**无限**顺序的有序流（infinite sequential ordered stream），它由迭代应用到初始元素种子的函数 f 产生"。回顾一下范例 2.4 讨论的 UnaryOperator，它是一种函数式接口，其输入参数和输出类型相同。如果有办法根据当前值生成流的下一个值，iterate 方法将相当有用，如例 3-4 所示。

例 3-4　利用 Stream.iterate 方法创建流

```
List<BigDecimal> nums =
    Stream.iterate(BigDecimal.ONE, n -> n.add(BigDecimal.ONE) )
        .limit(10)
        .collect(Collectors.toList());
System.out.println(nums);
// 打印[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Stream.iterate(LocalDate.now(), ld -> ld.plusDays(1L))
    .limit(10)
    .forEach(System.out::println)
// 打印从当日开始之后的10天
```

第一段代码采用 BigDecimal 实例，从 1 开始递增。第二段代码采用 java.time 包新增的 LocalDate 类，从当日开始按天递增。由于生成的两个流都是无界的，需要通过中间操作 limit 加以限制。

Stream 接口还定义了静态工厂方法 generate，其签名为：

```
static <T> Stream<T> generate(Supplier<T> s)
```

generate 方法通过多次调用 Supplier 产生一个顺序的无序流（sequential, unordered stream）。在 Java 标准库中，Supplier 的一种简单应用是 Math.random 方法，它不传入参数而返回 double 型数据，如例 3-5 所示。

例 3-5　利用 Math.random 创建随机流（double 型）

```
long count = Stream.generate(Math::random)
    .limit(10)
    .forEach(System.out::println)
```

如果已有集合，可以利用 Collection 接口新增的默认方法 stream，如例 3-6 所示。[1]

例 3-6　从集合创建流

```
List<String> bradyBunch = Arrays.asList("Greg", "Marcia", "Peter", "Jan",
    "Bobby", "Cindy");
names = bradyBunch.stream()
    .collect(Collectors.joining(","));
System.out.println(names);
// 打印Greg,Marcia,Peter,Jan,Bobby,Cindy
```

注 1：希望承认以下事实不会让我的声誉毁于一旦：我随口就能叫出《脱线家族》中六个孩子的姓名。真的，我也没想到自己的记忆力会这么好。

Stream 接口定义了三种专门用于处理基本数据类型的子接口，它们是 IntStream、LongStream 和 DoubleStream。IntStream 和 LongStream 还包括另外两种创建流所用的工厂方法 range 和 rangeClosed，二者的方法签名如下：

```
static IntStream  range(int startInclusive, int endExclusive)
static IntStream  rangeClosed(int startInclusive, int endInclusive)
static LongStream range(long startInclusive, long endExclusive)
static LongStream rangeClosed(long startInclusive, long endInclusive)
```

注意这几个语句中的参数有所不同：rangeClosed 包含终值（end value），而 range 不包含终值。两种方法都返回一个顺序的有序流，从第一个参数开始逐一递增。例 3-7 展示了 range 和 rangeClosed 方法的应用。

例 3-7　range 和 rangeClosed 方法

```
List<Integer> ints = IntStream.range(10, 15)
    .boxed()  ❶
    .collect(Collectors.toList());
System.out.println(ints);
// 打印[10, 11, 12, 13, 14]

List<Long> longs = LongStream.rangeClosed(10, 15)
    .boxed() ❶
    .collect(Collectors.toList());
System.out.println(longs);
// 打印[10, 11, 12, 13, 14, 15]
```

❶ Collectors 需要将基本数据类型转换为 List<T>

本例唯一的奇怪之处在于使用了 boxed 方法将 int 值转换为 Integer 实例。有关 boxed 方法的详细讨论请参见范例 3.2。

创建流所用的方法总结如下。

- Stream.of(T... values) 和 Stream.of(T t)
- Arrays.stream(T[] array) 以及用于处理 int[]、double[] 与 long[] 型数组的重载形式
- Stream.iterate(T seed, UnaryOperator<T> f)
- Stream.generate(Supplier<T> s)
- Collection.stream()
- 使用 range 和 rangeClosed 方法：
  - IntStream.range(int startInclusive, int endExclusive)
  - IntStream.rangeClosed(int startInclusive, int endInclusive)
  - LongStream.range(long startInclusive, long endExclusive)
  - LongStream.rangeClosed(long startInclusive, long endInclusive)

另见

流的应用贯穿全书。有关将基本类型流转换为包装器实例（wrapper instance）的讨论请参见范例 3.2。

3.2 装箱流

问题

用户希望利用基本类型流（primitive stream）创建集合。

方案

既可以使用 `java.util.stream.IntStream` 接口定义的 `boxed` 方法来包装元素，也可以使用合适的包装器类（wrapper class）来映射值，还可以使用 `collect` 方法的三参数形式。

讨论

在处理对象流（object stream）时，可以通过 `Collectors` 类提供的某种静态方法将流转换为集合。例如，对于一个给定的字符串流，例 3-8 显示了如何创建 `List<String>`。

例 3-8　将字符串流转换为列表

```
List<String> strings = Stream.of("this", "is", "a", "list", "of", "strings")
    .collect(Collectors.toList());
```

然而，同样的过程并不适合处理基本类型流，例 3-9 中的代码无法编译。

例 3-9　将 `int` 流转换为 `Integer` 列表（无法编译）

```
IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(Collectors.toList()); // 无法编译
```

有三种替代方案可以解决这个问题。第一种方案是利用 `boxed` 方法，将 `IntStream` 转换为 `Stream<Integer>`，如例 3-10 所示。

例 3-10　使用 `boxed` 方法

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .boxed() ❶
    .collect(Collectors.toList());
```

❶ 将 `int` 转换为 `Integer`

第二种方案是利用 `mapToObj` 方法，将基本类型流中的每个元素转换为包装类的一个实例，如例 3-11 所示。

例 3-11　使用 `mapToObj` 方法

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .mapToObj(Integer::valueOf)
    .collect(Collectors.toList())
```

`mapToInt`、`mapToLong` 和 `mapToDouble` 方法将对象流解析为相关的基本类型流，与之类似，`IntStream`、`LongStream` 与 `DoubleStream` 中的 `mapToObj` 方法将基本类型流转换为相关包装类的实例。在本例中，`mapToObj` 方法的参数为 `Integer` 类定义的静态方法 `valueOf`。



出于性能方面的考虑，构造函数 Integer(int val) 已被 JDK 9 弃用，建议改用 Integer.valueOf(int)。

第三种方案是采用 collect 方法的三参数形式，其签名为：

```
<R> R collect(Supplier<R> supplier,
              ObjIntConsumer<R> accumulator,
              BiConsumer<R,R> combiner)
```

collect 方法的用法如例 3-12 所示。

例 3-12　使用 collect 方法的三参数形式

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(ArrayList<Integer>::new, ArrayList::add, ArrayList::addAll);
```

如例所示，Supplier 是 ArrayList<Integer> 的构造函数。累加器（accumulator）为 add 方法，表示如何为列表添加单个元素。仅在并行操作中使用的组合器（combiner）是 addAll 方法，它能将两个列表合二为一。尽管 collect 的三参数形式并不常见，但理解其用法对开发很有好处。

以上三种方案均无不妥，采用哪种方案取决于开发人员的编程风格。

此外，如果希望将流转换为数组而非列表，那么采用 toArray 方法也不错，如例 3-13 所示。

例 3-13　将 IntStream 转换为 intArray

```
int[] intArray = IntStream.of(3, 1, 4, 1, 5, 9).toArray();
```

本范例讨论的三种方案都是必不可少的，这是 Java 最初将基本数据类型与对象区别对待的结果，因泛型的引入而变得复杂。不过，一旦掌握要领，使用 boxed 或 mapToObj 方法还是很容易的。

另见

有关收集器的讨论请参见第 4 章，有关构造函数引用的讨论请参见范例 1.3。

3.3　利用reduce方法实现归约操作

问题

用户希望通过流操作生成单一值。

方案

使用 reduce 方法对每个元素进行累加计算。

讨论

Java 的函数式范式经常采用"映射－筛选－归约"（map-filter-reduce）的过程处理数据。首先，map 操作将一种类型的流转换为另一种类型（如通过调用 length 方法将 String 流转换为 int 流）。接下来，filter 操作产生一个新的流，它仅包含所需的元素（如长度小于某个阈值的字符串）。最后，通过终止操作从流中生成单个值（如长度的总和或均值）。

1. 内置归约操作

基本类型流 IntStream、LongStream 和 DoubleStream 定义了多种内置在 API 中的归约操作。

例如，表 3-1 列出了 IntStream 接口定义的归约操作。

表3-1：IntStream接口定义的归约操作

方法	返回类型
average	OptionalDouble
count	long
max	OptionalInt
min	OptionalInt
sum	int
summaryStatistics	IntSummaryStatistics
collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R,R> combiner)	R
reduce	int, OptionalInt

sum、count、max、min、average 等归约操作的用途不言自明。有趣的是，如果流中没有元素（如经过筛选操作后），结果为空或未定义，以上提到的某些方法将返回 Optional。

例 3-14 显示了处理字符串集合长度的归约操作。

例 3-14　IntStream 接口的归约操作

```
String[] strings = "this is an array of strings".split(" ");
long count = Arrays.stream(strings)
        .map(String::length)          ❶
        .count();
System.out.println("There are " + count + " strings");

int totalLength = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .sum();
System.out.println("The total length is " + totalLength);

OptionalDouble ave = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .average();
System.out.println("The average length is " + ave);

OptionalInt max = Arrays.stream(strings)
        .mapToInt(String::length)     ❷
        .max();                       ❸
```

```
OptionalInt min = Arrays.stream(strings)
        .mapToInt(String::length)      ❷
        .min();                        ❸

System.out.println("The max and min lengths are " + max + " and " + min);
```

❶ count 是 Stream 接口定义的一种方法，因此无须将其映射给 IntStream

❷ sum 和 average 方法仅用于处理基本类型流

❸ 不带 Comparator 的 max 和 min 方法仅用于处理基本类型流

上述程序的打印结果如下：

```
There are 6 strings
The total length is 22
The average length is OptionalDouble[3.6666666666666665]
The max and min lengths are OptionalInt[7] and OptionalInt[2]
```

注意，average、max 与 min 方法返回 Optional，因为原则上可以通过应用一个筛选器来删除流中的所有元素。

count 方法相当有趣，相关讨论请参见范例 3.7。

Stream 接口定义了 max(Comparator) 和 min(Comparator) 方法，其中比较器用于确定最大元素和最小元素。而在 IntStream 接口中，由于比较操作采用整数的自然顺序完成，两种方法的重载形式均不需要参数。

有关 summaryStatistics 方法的讨论请参见范例 3.8。

表 3-1 中列出的最后两种归约操作 collect 和 reduce 值得进一步讨论。collect 方法的应用贯穿全书，其作用是将流转换为集合，通常与 Collectors 类定义的某种静态辅助方法配合使用（如 toList 或 toSet）。但是，无法在基本类型流中使用 collect 方法的三参数形式，即传入三个参数，分别是用于填充的集合、为集合添加单个元素的累加器以及为集合添加多个元素的组合器。有关这种形式的讨论请参见范例 3.2。

2. 基本归约实现

在实际接触到 reduce 方法之前，这种方法看起来可能不太直观。

IntStream 接口定义了 reduce 方法的两种重载形式：

```
OptionalInt reduce(IntBinaryOperator op)
int         reduce(int identity, IntBinaryOperator op)
```

第一条语句传入 IntBinaryOperator 并返回 OptionalInt，第二条语句需要提供 identity（int 型）以及 IntBinaryOperator。

读者或许还记得 java.util.function.BiFunction 接口，它传入两个参数并返回一个值，三者的类型可以不同。如果输入类型和返回类型相同，则函数为 BinaryOperator（如 Math.max）。注意，IntBinaryOperator 属于 BinaryOperator，其输入和输出类型均为 int。

那么，在不使用 sum 的情况下，如何实现整数的求和呢？一种方案是利用 reduce 方法，如例 3-15 所示。

例 3-15　利用 reduce 方法求和

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + y).orElse(0);  ❶
```

❶ sum 的值为 55



编写代码时，通常采用垂直方式安排流的流水线（stream pipeline），这是基于**流畅**（fluent）API 的一种方案，其中一个方法的结果将作为下一个方法的目标。在本例中，因为 reduce 方法返回的不是流，所以将 orElse 置于同一行（而非另起一行），它不属于流水线的一部分。不过这只是为了方便起见，读者可以根据需要使用任何格式。

在本例中，IntBinaryOperator 由 lambda 表达式提供，它传入两个 int 型数据并返回二者之和。不难想象，如果为 IntBinaryOperator 添加一个筛选器，流是可以为空的，其结果是 OptionalInt。之后的 orElse 方法表明，如果流中没有元素，返回值应该为 0。

在 lambda 表达式中，可以将二元运算符的第一个参数视为累加器，第二个参数视为流中每个元素的值。通过逐一打印各个元素能很容易理解这一点，如例 3-16 所示。

例 3-16　打印 *x* 和 *y* 的值

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> {
        System.out.printf("x=%d, y=%d%n", x, y);
        return x + y;
    }).orElse(0);
```

输出如例 3-17 所示。

例 3-17　逐一打印每个值的输出

```
x=1, y=2
x=3, y=3
x=6, y=4
x=10, y=5
x=15, y=6
x=21, y=7
x=28, y=8
x=36, y=9
x=45, y=10

sum=55
```

观察以上输出可知，*x* 和 *y* 的初始值是范围内的前两个值。二元运算符返回的值在下一次迭代时变为 *x*（累加器）的值，而 *y* 依次传入流的每个值。

那么，如果我们希望先处理每个数字，然后再求和呢？例如，在求和之前将所有的数字增加一倍 [2]。我们可能会写出如例 3-18 所示的代码，不过代码看似正确，实则有误。

注 2：可以采用多种方式解决这个问题，包括将 sum 方法返回的值增加一倍。这里介绍的方案演示了如何使用双参数形式的 reduce 方法。

例 3-18　在求和过程中将值增加一倍（代码错误）

```
int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + 2 * y).orElse(0);  ❶
```

❶ `doubleSum` 的值为 109（少了 1）

从 1 到 10 的各个整数之和为 55，因此增加一倍后的值应为 110，但本例的计算结果却是 109。问题出在 `reduce` 方法的 lambda 表达式上：x 和 y 的初始值为 1 和 2（流的前两个值）。换言之，流的第一个值不会增加一倍。

可以采用 `reduce` 方法的重载形式解决这个问题，也就是为累加器传入一个初始值。正确的代码如例 3-19 所示。

例 3-19　在求和过程中将值倍增（代码正确）

```
int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce(0, (x, y) -> x + 2 * y);  ❶
```

❶ `doubleSum` 的值为 110（这才是正确的值）

通过将累加器 x 的初始值设置为 0，y 的值被赋给流中的各个元素，从而实现所有元素增加一倍。例 3-20 显示了每次迭代时 x 和 y 的值。

例 3-20　每次迭代时 lambda 参数的值

```
Acc=0, n=1
Acc=2, n=2
Acc=6, n=3
Acc=12, n=4
Acc=20, n=5
Acc=30, n=6
Acc=42, n=7
Acc=56, n=8
Acc=72, n=9
Acc=90, n=10

sum=110
```

注意，当使用具有累加器初始值的 `reduce` 方法时，返回类型是 `int` 而非 `OptionalInt`。

二元运算符的标识值

本范例中的示例将第一个参数称为累加器的初始值（initial value），不过方法签名将其称为标识值（identity value）。关键字 `identity` 表示应该为二元运算符提供一个值，以便与其他值结合时返回另一个值。加法操作的标识值为 0，乘法操作的标识值为 1，字符串拼接操作的标识值为空字符串。

本节讨论的求和操作并无不同，但需要注意的是，应将计划用作二元运算符的任何操作的标识值作为 `reduce` 方法的第一个参数，即为累加器内部的初始值。

Java 标准库提供了多种归约方法，但如果这些方法都无法直接解决开发中遇到的问题，不妨试试本节讨论的两种 reduce 方法。

3. Java标准库中的二元运算符

标准库引入的一些新方法使归约操作变得特别简单。例如，Integer、Long 和 Double 类都定义了 sum 方法，其作用就是对两数求和。Integer 类中 sum 方法的实现如下所示：

```
public static int sum(int a, int b) {
    return a + b;
}
```

那么，为什么要专门定义一种只为实现两个整数求和的方法呢？这是因为 sum 方法属于 BinaryOperator（更确切地说，属于 IntBinaryOperator），很容易就能用于 reduce 方法，如例 3-21 所示。

例 3-21　利用二元运算符执行归约操作

```
int sum = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                .reduce(0, Integer::sum);
System.out.println(sum);
```

可以看到，无须使用 IntStream 就能得到相同的结果。Integer 类还定义了 max 和 min 方法，它们也是二元运算符，用法与 sum 方法类似，如例 3-22 所示。

例 3-22　利用 reduce 方法查找最大值

```
Integer max = Stream.of(3, 1, 4, 1, 5, 9)
        .reduce(Integer.MIN_VALUE, Integer::max); ❶
System.out.println("The max value is " + max);
```

❶ max 的标识值为最小的整数

另一个有趣的例子是 String 类定义的 concat 方法，它仅传入一个参数，看起来不怎么像二元运算符。

```
String concat(String str)
```

concat 方法可以用于 reduce 方法，如例 3-23 所示。

例 3-23　利用 reduce 方法拼接流中的字符串

```
String s = Stream.of("this", "is", "a", "list")
        .reduce("", String::concat);
System.out.println(s);     ❶
```

❶ 打印 thisisalist

上述代码之所以能执行，是因为通过类名（如 String::concat）使用方法引用时，第一个参数将作为 concat 的目标，而第二个参数是 concat 的参数。由于结果返回的是 String，目标、参数与返回类型均为同一类型，可以将其视为 reduce 方法的二元运算符。

concat 方法能大大缩减代码的尺寸，浏览 API 时请谨记在心。

使用收集器

尽管 concat 方法可行，但效率很低，因为字符串拼接操作会频繁创建和销毁对象。更好的方案是采用带有 Collector 的 collect 方法。

Stream 接口定义了 collect 方法的一种重载形式，它传入三个参数，分别是用于创建集合的 Supplier，为集合添加单个元素的 BiConsumer 以及合并两个集合的 BiConsumer。对字符串而言，StringBuilder 是一种天然的累加器。相应的 collect 实现如例 3-24 所示。

例 3-24　利用 StringBuilder 收集字符串

```
String s = Stream.of("this", "is", "a", "list")
        .collect(() -> new StringBuilder(),     ❶
                (sb, str) -> sb.append(str),    ❷
                (sb1, sb2) -> sb1.append(sb2))  ❸
        .toString();
```

❶ 结果 Supplier

❷ 为结果添加一个值

❸ 合并两个结果

可以通过方法引用简化上述代码，如例 3-25 所示。

例 3-25　利用方法引用收集字符串

```
String s = Stream.of("this", "is", "a", "list")
        .collect(StringBuilder::new,
                StringBuilder::append,
                StringBuilder::append)
        .toString();
```

不过，最简单的方案是采用 Collectors 工具类定义的 joining 方法，如例 3-26 所示。

例 3-26　利用 Collectors.joining 连接字符串

```
String s = Stream.of("this", "is", "a", "list")
        .collect(Collectors.joining());
```

joining 方法的重载形式传入字符串定界符，其简单易行无出其右。相关讨论请参见范例 4.2。

4. reduce方法的最一般形式

reduce 方法的第三种形式如下：

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

这种形式略显复杂，通常可以采用更简单的手段实现相同的目标。我们以一个示例说明这种形式的应用。

例 3-27 定义了一个 Book 类，它只有一个 ID（整数）和一个标题（字符串）。

例 3-27　简单的 Book 类

```
public class Book {
    private Integer id;
    private String title;

    // 构造函数、getter和setter、toString、equals、hashCode…
}
```

假设存在一个图书列表，我们希望将列表中的图书添加到某个 Map。其中键为 ID，值为图书本身。



采用 Collectors.toMap 方法解决这个问题更容易，相关讨论请参见范例 4.3。之所以以此为例，是因为它比较简单，有助于读者理解相对复杂的 reduce 方法。

例 3-28 显示了一种解决方案。

例 3-28　将 Book 添加到 Map

```
HashMap<Integer, Book> bookMap = books.stream()
    .reduce(new HashMap<Integer, Book>(),  ❶
           (map, book) -> {                ❷
                map.put(book.getId(), book);
                return map;
            },
            (map1, map2) -> {              ❸
                map1.putAll(map2);
                return map1;
            });

bookMap.forEach((k,v) -> System.out.println(k + ": " + v));
```

❶ putAll 的标识值

❷ 利用 put 将一本书添加到 Map

❸ 利用 putAll 合并多个 Map

我们从 reduce 方法的最后一个参数开始分析，这是最简单的。

第三个参数是 combiner，它必须是 BinaryOperator。在本例中，提供的 lambda 表达式传入两个映射，它将第二个映射中的所有键复制到第一个映射，再返回第一个映射。如果 putAll 方法能返回映射，lambda 表达式会更简单，可惜事实并非如此。仅当 reduce 方法并行完成时，组合器才有意义，因为我们需要将范围内每一部分产生的映射合并在一起。

第二个参数是一个函数，用于将一本书添加到 Map。类似地，如果 Map 的 put 方法在新条目添加完毕后能返回 Map，函数会更简单。

第一个参数是 combiner 函数的标识值。在本例中，标识值是一个为空的 Map，因为该标识值与其他任何 Map 结合后返回的是其他 Map。

例 3-28 的输出如下：

```
1: Book{id=1, title='Modern Java Recipes'}
2: Book{id=2, title='Making Java Groovy'}
3: Book{id=3, title='Gradle Recipes for Android'}
```

归约操作是函数式编程习惯用法的基础。在不少常见的用例中，`Stream` 接口都提供了相应的内置方法，如 `sum` 或 `collect(Collectors.joining(','))`。本范例也讨论了 `reduce` 方法的直接应用，或许能对读者编写自定义方法有所启发。

一旦掌握 Java 8 中 `reduce` 方法的用法，读者就能举一反三，理解如何在其他语言中使用相同的操作。即便这种操作被冠以不同的名称（如 Groovy 将其称为 `inject`，Scala 将其称为 `fold`），其原理并无差别。

另见

有关将 POJO 列表转换为 `Map` 的简便方案请参见范例 4.3，有关汇总统计（summary statistics）的讨论请参见范例 3.8，有关收集器的讨论请参见第 4 章。

3.4 利用reduce方法校验排序

问题

用户希望检查排序是否正确。

方案

使用 `reduce` 方法检查每对元素。

讨论

`java.util.stream.Stream` 接口定义的 `reduce` 方法传入 `BinaryOperator` 作为参数：

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

`BinaryOperator` 是一种输入类型和输出类型相同的 `Function`。根据范例 3.3 的讨论，`BinaryOperator` 的第一个元素通常为累加器，第二个元素传入流的每个值，如例 3-29 所示。

例 3-29　利用 `reduce` 方法对 `BigDecimal` 求和

```
BigDecimal total = Stream.iterate(BigDecimal.ONE, n -> n.add(BigDecimal.ONE))
        .limit(10)
        .reduce(BigDecimal.ZERO, (acc, val) -> acc.add(val)); ❶
System.out.println("The total is " + total);
```

❶ 使用 `BigDecimal` 类定义的 `add` 方法作为 `BinaryOperator`

与之前一样，lambda 表达式返回的任何值都将作为下一次迭代时变量 `acc` 的值。在本例中，程序计算前 10 个 `BigDecimal` 实例的值。

这是 reduce 方法最典型的应用方式。虽然 acc 在本例中用作累加器，但并不意味着它必须作为累加器使用。接下来，我们采用范例 4.1 讨论的方法来排序字符串。例 3-30 展示的代码段根据字符串的长度对它们进行排序。

例 3-30　根据长度对字符串排序

```
List<String> strings = Arrays.asList(
    "this", "is", "a", "list", "of", "strings");

List<String> sorted = strings.stream()
    .sorted(Comparator.comparingInt(String::length))
    .collect(toList());    ❶
```

❶ 结果为 ["a", "is", "of", "this", "list", "strings"]

那么，如何验证排序是否正确呢？答案是比较每对相邻的字符串，确保第一个字符串的长度不大于第二个字符串。reduce 方法就能实现这个功能，如例 3-31 所示（Junit 测试用例的一部分）。

例 3-31　测试字符串排序是否正确

```
strings.stream()
    .reduce((prev, curr) -> {
        assertTrue(prev.length() <= curr.length()); ❶
        return curr;                                ❷
    });
```

❶ 检查每对字符串的排序是否正确

❷ curr 成为 prev 的下一个值

对于每对连续的字符串，程序将前一个参数和当前参数分别赋给变量 prev 和 curr。assertTrue 用于测试前一个字符串的长度是否小于或等于当前字符串的长度。需要注意的是，reduce 方法的参数将返回当前字符串 curr 的值，它在下一次迭代时成为 prev 的值。

为使上述代码能正确执行，唯一的要求是采用顺序（sequential）流或有序流。

另见

有关 reduce 方法的讨论请参见范例 3.3，有关排序的讨论请参见范例 4.1。

3.5　利用peek方法对流进行调试

问题

用户希望在处理流时查看流中的各个元素。

方案

根据需要，在流的流水线中调用 java.util.stream.Stream 接口定义的中间操作 peek。

讨论

流处理由一系列零个或多个中间操作构成，后跟终止操作。每个中间操作都返回一个新的流，而终止操作将返回不是流的值。

对于流的流水线所涉及的中间操作序列，初次接触 Java 8 的用户有时候会感到困惑，因为无法在处理流时查看各个元素的值。

我们考虑这样一个简单的方法，即接受某个整数流的开始范围和结束范围，并将每个数字倍增，然后仅对能被 3 整除的结果值求和，如例 3-32 所示。

例 3-32　对整数进行倍增、筛选与求和

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

编写一个简单的测试，验证程序可以正确执行：

```
@Test
public void sumDoublesDivisibleBy3() throws Exception {
    assertEquals(1554, demo.sumDoublesDivisibleBy3(100, 120));
}
```

上述测试虽然有一定帮助，但并未提供太多有价值的信息。如果代码无法运行，很难找出症结所在。

如例 3-33 所示，我们为流水线添加一个 `map` 操作，传入并打印每个值，然后再次返回这些值。

例 3-33　添加标识映射以便打印

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> { ❶
            System.out.println(n);
            return n;
        })
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

❶ 标识映射（identity map）打印并返回每个元素

程序将打印从 `start`（含）到 `end`（含）的数字，每行一个数字。尽管不应在生产环境中使用上述代码，但它的确能让用户在不影响流处理的同时观察其内部操作。

这正是 `Stream` 接口中 `peek` 方法的工作原理，该方法的声明如下：

```
Stream<T> peek(Consumer<? super T> action)
```

根据 Javadoc 的描述，`peek` 方法“返回一个由流的元素构成的流，当元素从所生成的流中

消耗时，对每个元素执行给定的操作”。由于 Consumer 仅传入一个输入而不返回任何值，所提供的 Consumer 不会对值造成破坏。

peek 方法是一种中间操作，可以根据需要多次添加，如例 3-34 所示。

例 3-34　使用多个 peek 方法

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .peek(n -> System.out.printf("original: %d%n", n)) ❶
        .map(n -> n * 2)
        .peek(n -> System.out.printf("doubled : %d%n", n)) ❷
        .filter(n -> n % 3 == 0)
        .peek(n -> System.out.printf("filtered: %d%n", n)) ❸
        .sum();
}
```

❶ 在倍增前打印值

❷ 在倍增后、筛选前打印值

❸ 在筛选后、求和前打印值

程序将显示每个元素的初始值、倍增值以及筛选后的值，输出结果如下：

```
original: 100
doubled : 200
original: 101
doubled : 202
original: 102
doubled : 204
filtered: 204
...
original: 119
doubled : 238
original: 120
doubled : 240
filtered: 240
```

略显不便的是，很难将用于测试的 peek 方法设置为可选。因此，尽管 peek 方法有助于调试，但不应将其置于生产环境中。

3.6　字符串与流之间的转换

问题

用户希望通过惯用的流处理技术（而不是对 String 中的各个字符进行循环）实现字符串与流之间的转换。

方案

使用 java.lang.CharSequence 接口定义的默认方法 chars 和 codePoints，将 String 转换

为 IntStream。为了将 IntStream 转换回 String，使用 java.util.stream.IntStream 接口定义的 collect 方法的重载形式。它传入三个参数，分别是 Supplier、表示累加器的 BiConsumer 以及表示组合器的 BiConsumer。

讨论

字符串是若干字符的集合。理论上说，将字符串转换为流并不困难，如同将字符串转换为集合或数组一样。遗憾的是，String 不属于集合框架（collections framework），因此无法实现 Iterable，不存在一种能将 String 转换为 Stream 的 stream 工厂方法。另一种方案是采用 java.util.Array 类定义的各种静态 stream 方法。然而，尽管 Arrays.stream 提供了用于处理 int[]、long[]、double[] 甚至 T[] 的方法，却并未定义用于处理 char[] 的方法。API 的设计者似乎不希望用户采用流技术处理字符串。

尽管如此，仍然有办法实现字符串与流之间的转换。String 类实现 CharSequence 接口，它引入了两种能生成 IntStream 的方法（chars 和 codePoints），它们都是接口中的默认方法，因此存在可用的实现。例 3-35 展示了两种方法的签名。

例 3-35　CharSequence 接口定义的 chars 和 codePoints 方法

```
default IntStream chars()
default IntStream codePoints()
```

chars 和 codePoints 方法的不同之处在于，chars 方法用于处理 UTF-16 编码字符，而 codePoints 方法用于处理完整的 Unicode 代码点（code point）集。如果读者对两种方法之间的差异感兴趣，可以阅读 Javadoc 中有关 java.lang.Character 类的描述。就本范例而言，区别只在于返回的整数类型：chars 方法返回一个由序列中的 char 值构成的 IntStream，而 codePoints 方法返回一个由 Unicode 代码点构成的 IntStream。

那么，如何将字符流转换回字符串呢？Stream.collect 方法对流元素执行可变归约（mutable reduction）操作以生成集合。Collectors 工具类提供了大量可以生成所需 Collector 的静态方法（如本书讨论的 toList、toSet、toMap、joining 以及其他许多方法），因此传入 Collector 的 collect 方法在开发中最为常用。

然而，明显看出缺少的是 Collector 传入一个字符流并将其组装为字符串。好在代码并不复杂，可以使用 collect 的另一种重载形式，它传入一个 Supplier 以及两个分别作为累加器和组合器的 BiConsumer 参数。

听起来似乎比实际情况要复杂得多。接下来，我们编写 isPalindrome 方法，以检查某个字符串是否属于回文（palindrome）。回文检查器不区分大小写，它首先删除结果字符串中存在的标点符号，再检查字符串是否正读和反读都一样。用于测试字符串的 isPalindrome 方法如例 3-36 所示，这是 Java 7 及之前版本的实现。

例 3-36　检查字符串是否属于回文（Java 7 及之前）

```
public boolean isPalindrome(String s) {
    StringBuilder sb = new StringBuilder();
    for (char c : s.toCharArray()) {
        if (Character.isLetterOrDigit(c)) {
            sb.append(c);
```

```
        }
    }
    String forward = sb.toString().toLowerCase();
    String backward = sb.reverse().toString().toLowerCase();
    return forward.equals(backward);
}
```

以上代码具有典型的非函数式编程风格。isPalindrome 方法首先声明一个具有可变状态的单独对象（StringBuilder 实例），然后对集合进行迭代（由 String 类定义的 toCharArray 方法返回的 char[]），并利用 if 条件语句决定是否将值附加到缓冲区。StringBuilder 类还定义了一个能更容易实现回文检查的 reverse 方法，String 类则没有类似的方法。这种可变状态、迭代、决策语句的组合迫切需要一种基于流的替代方案，如例 3-37 所示，这是 Java 8 的实现。

例 3-37　检查字符串是否属于回文（Java 8）

```
public boolean isPalindrome(String s) {
    String forward = s.toLowerCase().codePoints() ❶
        .filter(Character::isLetterOrDigit)
        .collect(StringBuilder::new,
                 StringBuilder::appendCodePoint,
                 StringBuilder::append)
        .toString();

  String backward = new StringBuilder(forward).reverse().toString();
  return forward.equals(backward);
}
```

❶ 返回 IntStream

在本例中，codePoints 方法返回 IntStream，之后可以使用与例 3-37 相同的条件进行筛选。有意思的是 collect 方法，其签名为：

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

这三个参数的用途如下。

- Supplier 生成经过归约的对象（本例为 StringBuilder）。
- 第一个 BiConsumer 将流的各个元素累加至所生成的数据结构，本例使用 appendCodePoint 方法。
- 第二个 BiConsumer 表示组合器，它是一个“无干扰的无状态函数”（non-interfering, stateless function），用于将两个必须与累加器兼容的值组合在一起（本例为 append 方法）。注意，组合器仅在并行操作时使用。

collect 方法的参数略多，不过其优点在于代码不必区分字符和整数，而这是处理字符串元素时经常遇到的问题。

例 3-38 显示了针对回文检查器的简单测试。

例 3-38　测试回文检查器

```java
private PalindromeEvaluator demo = new PalindromeEvaluator();

@Test
public void isPalindrome() throws Exception {
    assertTrue(
        Stream.of("Madam, in Eden, I'm Adam",
                  "Go hang a salami; I'm a lasagna hog",
                  "Flee to me, remote elf!",
                  "A Santa pets rats as Pat taps a star step at NASA")
              .allMatch(demo::isPalindrome));

    assertFalse(demo.isPalindrome("This is NOT a palindrome"));
}
```

将字符串视为一种字符数组不太符合 Java 8 倡导的函数式习惯用法，但希望本范例讨论的机制能对读者有所启发。

另见

有关收集器的讨论请参见第 4 章，有关实现自定义收集器的讨论请参见范例 4.9，有关 `allMatch` 方法的讨论请参见范例 3.10。

3.7　获取元素数量

问题

用户希望获取流中元素的数量。

方案

使用 `java.util.stream.Stream` 接口定义的 `count` 方法，或 `java.util.stream.Collectors` 类定义的 `counting` 方法。

讨论

本范例相当简单，目的是为范例 4.6 讨论的下游收集器作铺垫。

如例 3-39 所示，`Stream` 接口定义了 `count` 默认方法，它能返回 `long` 型数据。

例 3-39　利用 `Stream.count` 方法获取元素数量

```java
long count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5).count();
System.out.printf("There are %d elements in the stream%n", count); ❶
```

❶ 打印 `There are 9 elements in the stream`

`count` 方法的有趣之处在于它的实现方式。根据 Javadoc 的描述，“这是一种特殊的归约操作，相当于”：

```
return mapToLong(e -> 1L).sum();
```

首先，流的每个元素都被映射为 1（long）。然后，mapToLong 方法生成 LongStream，它定义了 sum 方法。换言之，先将所有元素映射为 1，再将它们相加，简单明了。

此外，Collectors 类定义了一种类似的方法 counting，如例 3-40 所示。

例 3-40　利用 Collectors.counting 方法获取元素数量

```
count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .collect(Collectors.counting());
System.out.printf("There are %d elements in the stream%n", count);
```

这两种方法得到的结果并无不同，但既然已有 Stream.count，为什么还要讨论 Collectors.counting 呢？

我们当然可以使用 Stream.count 方法，按说也应该这样处理。不过“下游收集器”（downstream collector）的使用将在范例 4.6 进行详细讨论。就目前而言，我们考虑例 3-41。

例 3-41　对根据长度划分的字符串计数

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(
        s -> s.length() % 2 == 0, ❶
        Collectors.counting()));  ❷

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n", k, v));
//
// false: 4
//  true: 8
```

❶ 谓词

❷ 下游收集器

partitioningBy 方法的第一个参数是 Predicate，其作用是将字符串分为满足谓词和不满足谓词的两类。如果 partitioningBy 方法只有这一个参数，则结果为 Map<Boolean, List<String>>，其中键为 true 和 false，值为偶数长度和奇数长度字符串的列表。

本例采用 partitioningBy 方法的双参数重载形式，它传入 Predicate 和 Collector。Collector 被称为下游收集器，用于对返回的每个字符串列表进行后期处理。这就是 Collectors.counting 方法的用例。双参数形式的 partitioningBy 方法将输出 Map<Boolean, Long>，其值为流中偶数长度和奇数长度字符串的数量。

Stream 接口定义的其他几种方法在 Collectors 类中也有对应的方法，其他章节将对它们进行讨论。简而言之，直接处理流时请使用 Stream 定义的方法，而 Collectors 定义的方法适用于 partitioningBy 或 groupingBy 操作的下游后期处理。

另见

有关下游收集器的讨论请参见范例 4.6，收集器的一般性介绍请参见第 4 章，有关归约操作的讨论请参见范例 3.3。

3.8 汇总统计

问题

用户希望获取数值流中元素的数量、总和、最小值、最大值以及平均值。

方案

使用 `IntStream`、`DoubleStream` 或 `LongStream` 接口定义的 `summaryStatistics` 方法。

讨论

基本类型流 `IntStream`、`DoubleStream` 与 `LongStream` 为 `Stream` 接口引入了用于处理基本数据类型的方法，`summaryStatistics` 就是其中一种方法，如例 3-42 所示。

例 3-42 `summaryStatistics` 方法

```
DoubleSummaryStatistics stats = DoubleStream.generate(Math::random)
    .limit(1_000_000)
    .summaryStatistics();

System.out.println(stats);   ❶

System.out.println("count: " + stats.getCount());
System.out.println("min  : " + stats.getMin());
System.out.println("max  : " + stats.getMax());
System.out.println("sum  : " + stats.getSum());
System.out.println("ave  : " + stats.getAverage());
```

❶ 使用 `toString` 方法打印



从 Java 7 开始，可以在数字字面量中使用下划线，如 1_000_000。

执行上述程序，输出结果类似于：

```
DoubleSummaryStatistics{count=1000000, sum=499608.317465, min=0.000001,
    average=0.499608, max=0.999999}
count: 1000000
min  : 1.3938598313334438E-6
max  : 0.9999988915490642
sum  : 499608.31746475823
ave  : 0.49960831746475826
```

`DoubleSummaryStatistics` 类定义的 `toString` 方法返回字符串的表达形式，也提供用于统计元素数量、总和、最小值、最大值以及平均值的 getter 方法（`getCount`、`getSum`、`getMax`、`getMin` 以及 `getAverage`）。当 `double` 型元素的数量达到 100 万时，最小值趋近于 0，最大值趋近于 1，总和约为 50 万，平均值约为 0.5。

DoubleSummaryStatistics 类还定义了以下两个有趣的方法：

```
void accept(double value)
void combine(DoubleSummaryStatistics other)
```

accept 方法用于在汇总信息中记录另一个值，而 combine 方法将两个 DoubleSummaryStatistics 对象合二为一。两种方法在计算结果之前，向类的实例添加数据时使用。

我们以运动员薪资跟踪网站 Spotrac 为例进行讨论。本书的源代码包括一个文件，它记录了 2017 赛季美国职棒大联盟（MLB）全部 30 支球队的薪资数据，这些数据均来自 Spotrac。

例 3-43 定义了一个名为 Team 的类，包括 id、name（队名）与 salary（薪资）。

例 3-43　Team 类包括 id、name 与 salary

```
public class Team {
    private static final NumberFormat nf = NumberFormat.getCurrencyInstance();

    private int id;
    private String name;

    private double salary;

    // 构造函数、getter与setter

    @Override
    public String toString() {
        return "Team{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", salary=" + nf.format(salary) +
                '}';
    }
}
```

解析球队工资文件，结果如下：

```
Team{id=1, name='Los Angeles Dodgers', salary=$245,269,535.00}
Team{id=2, name='Boston Red Sox', salary=$202,135,939.00}
Team{id=3, name='New York Yankees', salary=$202,095,552.00}
...
Team{id=28, name='San Diego Padres', salary=$73,754,027.00}
Team{id=29, name='Tampa Bay Rays', salary=$73,102,766.00}
Team{id=30, name='Milwaukee Brewers', salary=$62,094,433.00}
```

可以通过两种方式计算球队集合的汇总统计信息。第一种方式采用 collect 方法的三参数形式，如例 3-44 所示。

例 3-44　传入 Supplier、累加器与组合器的 collect 方法

```
DoubleSummaryStatistics teamStats = teams.stream()
        .mapToDouble(Team::getSalary)
        .collect(DoubleSummaryStatistics::new,
                 DoubleSummaryStatistics::accept,
                 DoubleSummaryStatistics::combine);
```

有关这种 collect 方法的讨论请参见范例 4.9。在本例中，collect 方法通过构造函数引用来提供 DoubleSummaryStatistics 的实例，通过 accept 方法将另一个值添加到现有的 DoubleSummaryStatistics 对象，以及通过 combine 方法将两个单独的 DoubleSummaryStatistics 对象合二为一。

输出结果如下（为便于阅读，对结果做了处理）：

```
30 teams
  sum = $4,232,271,100.00
  min =    $62,094,433.00
  max =   $245,269,535.00
  ave =   $141,075,703.33
```

计算汇总信息的另一种方案请参见范例 4.6（下游收集器）。此时，汇总计算如例 3-45 所示。

例 3-45　使用 summarizingDouble 方法进行收集

```
teamStats = teams.stream()
        .collect(Collectors.summarizingDouble(Team::getSalary));
```

其中，Collectors.summarizingDouble 方法的参数是各队的薪资。无论采用哪种方案，结果并无区别。

从本质上讲，汇总统计类是一种“糟糕”的统计方法，因为它们仅能统计数量、最大值、最小值、总和、平均值等属性。然而，如果只需要这些属性，那么 Java 标准库应能满足需要。[3]

另见

汇总统计是归约操作的一种特殊形式，其他归约操作的介绍请参见范例 3.3，有关下游收集器的讨论请参见范例 4.6，有关多参数 collect 方法的讨论请参见范例 4.9。

3.9　查找流的第一个元素

问题

用户希望查找满足流中特定条件的第一个元素。

方案

应用筛选器之后使用 java.util.stream.Stream 接口定义的 findFirst 或 findAny 方法。

讨论

Stream 接口定义的 findFirst 方法返回描述流中第一个元素的 Optional，而 findAny 方法

注 3：当然，读者从本范例中还应学到一点：如果有机会参加 MLB 比赛，那么不妨一试，哪怕只有很短的上场时间。赛后再继续学习 Java 吧！

返回描述流中某个元素的 `Optional`。两种方法都不传入参数，意味着映射或筛选操作已经完成。

例如，给定一个整数列表，为查找第一个偶数，可以在应用偶数筛选器后使用 `findFirst` 方法，如例 3-46 所示。

例 3-46　查找第一个偶数

```
Optional<Integer> firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .filter(n -> n % 2 == 0)
    .findFirst();

System.out.println(firstEven); ❶
```

❶ 打印 `Optional[4]`

如果流为空，则返回值是一个空 `Optional`（如例 3-47 所示）。

例 3-47　流为空时使用 `findFirst` 方法

```
Optional<Integer> firstEvenGT10 = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .filter(n -> n > 10)
    .filter(n -> n % 2 == 0)
    .findFirst();

System.out.println(firstEvenGT10); ❶
```

❶ 打印 `Optional.empty`

上述代码在应用筛选器后返回第一个元素，这是否意味着要做大量无用功呢？为什么要对所有元素进行取模运算，然后只选择第一个元素呢？由于流元素实际上是逐一进行处理的，这不是一个问题，相关讨论请参见范例 3.13。

如果流不存在出现顺序（encounter order），它可能返回任何元素。不过在本例中，流确实具有出现顺序，因此**无论采用顺序流还是并行流进行搜索**，“第一个”偶数始终是 4。详见例 3-48。

例 3-48　在并行流中使用 `firstEven`

```
firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .parallel()
    .filter(n -> n % 2 == 0)
    .findFirst();

System.out.println(firstEven); ❶
```

❶ 始终打印 `Optional[4]`

初看之下有些奇怪，为什么在同时处理多个数字时仍然会得到同一个值呢？原因在于**出现顺序**的概念。

Java API 将出现顺序定义为数据源使其元素可用的顺序。`List` 和 `Array` 都有出现顺序，但 `Set` 没有。

`BaseStream` 接口（`Stream` 的父接口）还定义了一种名为 `unordered` 的方法，它可能返回（也可能不返回）一个无序流作为中间操作。

set 与出现顺序

虽然 HashSet 实例本身没有出现顺序，但如果重复初始化包含相同数据的 HashSet 实例（Java 8），则每次返回的元素顺序都相同，这意味着每次调用 findFirst 方法也会得到相同的结果。根据 Javadoc 的描述，findFirst 用于无序流时**可能**会获得不同的结果，但当前的实现不会因为流是无序的而改变其行为。

如果希望获得一个具有不同出现顺序的 Set，可以通过添加和删除足够多的元素来强制进行再散列操作（rehashing）。例如：

```
List<String> wordList = Arrays.asList(
    "this", "is", "a", "stream", "of", "strings");
Set<String> words = new HashSet<>(wordList);
Set<String> words2 = new HashSet<>(words);

// 接下来，添加和删除足够多的元素来强制进行再散列操作
IntStream.rangeClosed(0, 50).forEachOrdered(i ->
  words2.add(String.valueOf(i)));
words2.retainAll(wordList);

// 这些集合是相等的，但具有不同的元素排序
System.out.println(words.equals(words2));
System.out.println("Before: " + words);
System.out.println("After : " + words2);
```

输出结果类似于：

```
true
Before: [a, strings, stream, of, this, is]
After : [this, is, strings, stream, of, a]
```

可以看到，两次排序并不相同，因此调用 findFirst 的结果也将有所不同。

在 Java 9 中，新的不可变集合（与映射）是随机的，即使它们每次都以相同的方式初始化，其每次运行的迭代顺序也会发生变化。[4]

findAny 方法要么返回描述流中某个元素的 Optional，要么在流为空时返回一个空 Optional。在本例中，操作的行为具有**显式不确定性**（explicit nondeterminism），这意味着可以自由选择流中的任何元素，从而实现对并行操作的优化。

为说明这一点，考虑从一个无序的并行整数流中返回任意元素。例 3-49 在随机延迟 100 毫秒后将每个元素映射到其自身，从而引入了一个人为的延迟。

例 3-49　随机延迟后在并行流中使用 findAny 方法

```
public Integer delay(Integer n) {
    try {
        Thread.sleep((long) (Math.random() * 100));
    } catch (InterruptedException ignored) { ❶
```

注 4：感谢 Stuart Marks 所做的解释。

```
    }
    return n;
}

// 其他代码

Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()       ❷
    .parallel()        ❸
    .map(this::delay)  ❹
    .findAny();        ❺

System.out.println("Any: " + any);
```

❶ Java 中唯一可以捕获和忽略的异常[5]

❷ 顺序并不重要

❸ 在并行流中采用通用 fork/join 线程池

❹ 引入随机延迟

❺ 无论出现顺序如何，返回第一个元素

上述代码可以输出任何给定的数字，取决于先执行到哪个线程。

`findFirst` 和 `findAny` 都属于**短路终止**操作（short-circuiting, terminal operation）。当作用于无限流时，短路操作可能产生一个有限流。如果终止操作在作用于无限输入时也可能在有限时间内终止，它就属于短路操作。

从这一节讨论的示例可以看到，并行（parallelization）在某些情况下非但不会提高性能，反而会降低性能。流是惰性的，它们只处理满足流水线所需的元素。在本例中，由于只要求返回第一个元素，启动 fork/join 线程池显得有些矫枉过正。详见例 3-50。

例 3-50　在顺序流和并行流中使用 `findAny` 方法

```
Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .map(this::delay)
    .findAny(); ❶

System.out.println("Sequential Any: " + any);

any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .parallel()
    .map(this::delay)
    .findAny(); ❷

System.out.println("Parallel Any: " + any);
```

❶ 顺序流（默认）

注 5：严格来说，不应该捕获并忽略任何异常。尽管忽略 `InterruptedException` 的做法很常见，但这并不是个好主意。

❷ 并行流

典型的输出如下所示（在一台 8 核计算机上运行，默认使用 8 线程 fork/join 线程池）。[6]

对于顺序处理：

```
main // 顺序处理，因此只有一个线程
Sequential Any: Optional[3]
```

对于并行处理：

```
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-6
ForkJoinPool.commonPool-worker-7
main
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-4
Parallel Any: Optional[1]
```

顺序流只需访问并返回一个元素，因此属于短路操作。并行流启动 8 个不同的线程，找到一个元素后关闭所有线程。换言之，并行流访问了大量并不需要的值。

请注意，流的出现顺序是一个重要概念。如果流具有出现顺序，`findFirst` 方法总是会返回同一个值。而 `findAny` 方法可以返回任意元素，因此更适合在并行操作中使用。

另见

有关惰性流的讨论请参见范例 3.13，有关并行流的讨论请参见第 9 章。

3.10　使用anyMatch、allMatch与noneMatch方法

问题

用户希望确定流中是否有元素匹配 `Predicate`，或全部元素匹配 `Predicate`，或没有元素匹配 `Predicate`。

方案

使用 `java.util.stream.Stream` 接口定义的 `anyMatch`、`allMatch` 与 `noneMatch` 方法，每种方法返回一个布尔值。

讨论

`anyMatch`、`allMatch` 与 `noneMatch` 方法的签名如下：

注 6：这里假定已将延迟方法修改为打印当前线程的名称及其正在处理的值。

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

每种方法的用途不言自明。我们以质数计算器为例进行说明。在大于或等于 2 的自然数中，如果一个数无法被除 1 和该数自身之外的其他数整除，则这个数为质数（prime number），否则为合数（composite number）。

如例 3-51 所示，为验证某个数是否为质数，一种简单的做法是对从 2 开始到该数平方根的所有数做取模运算，然后取整。

例 3-51　质数校验

```
public boolean isPrime(int num) {
    int limit = (int) (Math.sqrt(num) + 1);         ❶
    return num == 2 || num > 1 && IntStream.range(2, limit)
        .noneMatch(divisor -> num % divisor == 0); ❷
}
```

❶ 校验上限

❷ 使用 noneMatch 方法

借由 noneMatch 方法，质数校验易如反掌。

> **BigInteger 类与质数**
>
> java.math.BigInteger 类定义的 isProbablePrime 方法很有意思，其签名为：
>
> ```
> boolean isProbablePrime(int certainty)
> ```
>
> 如果 isProbablePrime 方法返回 false，则值显然为合数；如果返回 true，certainty 参数就派上用场了。
>
> certainty 的值表示调用程序可以容忍的不确定性。如果 isProbablePrime 方法返回 true，则一个数实际为质数的概率将超过 1 - 1/2^{certainty}。因此，certainty 等于 2 意味着概率为 0.75，certainty 等于 3 意味着概率为 0.875，certainty 等于 4 意味着概率为 0.9375，等于 5 意味着概率为 0.96875，以此类推。
>
> certainty 参数的值越大，isProbablePrime 方法的执行时间越长。

例 3-52 显示了质数校验的两种方案。

例 3-52　针对质数计算的测试

```
private Primes calculator = new Primes();

@Test ❶
public void testIsPrimeUsingAllMatch() throws Exception {
    assertTrue(IntStream.of(2, 3, 5, 7, 11, 13, 17, 19)
        .allMatch(calculator::isPrime));
}

@Test ❷
```

```
    public void testIsPrimeWithComposites() throws Exception {
        assertFalse(Stream.of(4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20)
            .anyMatch(calculator::isPrime));
    }
```

❶ 为简单起见，使用 allMatch 方法

❷ 使用合数进行测试

第一个测试调用已知质数流中的 allMatch 方法（参数为 Predicate），仅当所有值均为质数时返回 true。

第二个测试对一个合数集合使用 anyMatch 方法，并认定集合中的数字均不满足谓词。

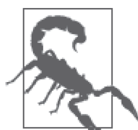
anyMatch、allMatch 与 noneMatch 方法能方便地对特定条件下的值流进行校验。

不过，有一个可能引起问题的边缘条件应予注意。如例 3-53 所示，三种方法在作用于空流（empty stream）时的行为不那么直观。

例 3-53　针对空流的测试

```
    @Test
    public void emptyStreamsDanger() throws Exception {
        assertTrue(Stream.empty().allMatch(e -> false));
        assertTrue(Stream.empty().noneMatch(e -> true));
        assertFalse(Stream.empty().anyMatch(e -> true));
    }
```

根据 Javadoc 的描述，对于 allMatch 和 noneMatch 方法，“流为空将返回 true 且不再评估谓词”，因此两种方法中的谓词可以是任何值。而对于 anyMatch 方法，流为空将返回 false，这可能导致错误诊断异常困难，所以应用时务须谨慎。



如果流为空，无论提供的谓词是什么，allMatch 和 noneMatch 方法将返回 true，而 anyMatch 方法将返回 false。三种方法在流为空时都不会评估任何提供的谓词。

另见

有关 Predicate 接口的讨论请参见范例 2.3。

3.11　使用flatMap与map方法

问题

用户希望以某种方式转换流中的元素，但不确定该使用 map 还是 flatMap 方法。

方案

如果需要将每个元素转换为一个值，则使用 Stream.map 方法；如果需要将每个元素转换为多个值，且需要将生成的流“展平”，则使用 Stream.flatMap 方法。

讨论

map 和 flatMap 方法均传入 Function 作为参数。map 方法的签名如下：

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Function 传入一个输入，并将其转换为一个输出。map 方法则将一个 T 类型的输入转换为一个 R 类型的输出。

我们创建一个由顾客名和 Order 集合构成的 Customer 类。为简单起见，Order 类只包含一个整数 ID。例 3-54 展示了 Customer 类和 Order 类。

例 3-54　一对多关系

```
public class Customer {
    private String name;
    private List<Order> orders = new ArrayList<>();

    public Customer(String name) {
        this.name = name;
    }

    public String getName() { return name; }
    public List<Order> getOrders() { return orders; }

    public Customer addOrder(Order order) {
        orders.add(order);
        return this;
    }
}

public class Order {
    private int id;

    public Order(int id) {
        this.id = id;
    }

    public int getId() { return id; }
}
```

接下来，我们创建若干新顾客并添加一些订单，如例 3-55 所示。

例 3-55　客户与订单示例

```
Customer sheridan = new Customer("Sheridan");
Customer ivanova = new Customer("Ivanova");
Customer garibaldi = new Customer("Garibaldi");

sheridan.addOrder(new Order(1))
        .addOrder(new Order(2))
        .addOrder(new Order(3));
ivanova.addOrder(new Order(4))
        .addOrder(new Order(5));

List<Customer> customers = Arrays.asList(sheridan, ivanova, garibaldi);
```

当输入参数和输出类型之间存在一一对应的关系时，将执行 `map` 操作。可以将顾客映射到他们的姓名并打印，如例 3-56 所示。

例 3-56　将顾客映射到他们的姓名

```
customers.stream()                    ❶
        .map(Customer::getName)       ❷
        .forEach(System.out::println); ❸
```

❶ `Stream<Customer>`

❷ `Stream<String>`

❸ 谢里登、伊万诺娃、加里波第

如果将顾客映射到订单而不是他们的姓名，就得到了一个集合的集合，如例 3-57 所示。

例 3-57　将顾客映射到订单

```
customers.stream()
        .map(Customer::getOrders)                     ❶
        .forEach(System.out::println);                ❷

customers.stream()
        .map(customer -> customer.getOrders().stream()) ❸
        .forEach(System.out::println);
```

❶ `Stream<List<Order>>`

❷ `[Order{id=1}, Order{id=2}, Order{id=3}], [Order{id=4}, Order{id=5}], []`

❸ `Stream<Stream<Order>>`

`map` 操作的结果为 `Stream<List<Order>>`，其最后一个列表为空。如果在订单列表中调用 `stream` 方法，则结果为 `Stream<Stream<Order>>`，其最后一个内部流（inner stream）为空流。

`flatMap` 方法的作用就在于此，其签名如下：

```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

对于每个泛型参数 `T`，函数生成的是 `Stream<R>` 而不仅仅是 `R`。之后，`flatMap` 方法从各个流中删除每个元素并将它们添加到输出，从而“展平”生成的流。



`flatMap` 方法的 `Function` 参数传入一个泛型输入参数，但生成的输出类型为 `Stream`。

`flatMap` 方法的应用如例 3-58 所示。

例 3-58　对顾客订单应用 `flatMap` 方法

```
customers.stream()                                      ❶
        .flatMap(customer -> customer.getOrders().stream()) ❷
        .forEach(System.out::println);                  ❸
```

❶ `Stream<Customer>`

❷ Stream<Order>

❸ Order{id=1}, Order{id=2}, Order{id=3}, Order{id=4}, Order{id=5}

flatMap 操作的结果为 Stream<Order>。由于它已被“展平”，无须再担心嵌套流（nested stream）。

与 flatMap 方法有关的两个重要概念应予注意：

- 方法参数 Function 产生一个输出值流；
- 生成的元素被“展平”为一个新的流。

将上述两点谨记在心，就能体会到 flatMap 方法的有用之处。

最后需要指出的是，Java 8 引入的 Optional 类同样定义了 map 和 flatMap 方法，详细讨论请参见范例 6.4 和范例 6.5。

另见

有关 Optional.map 方法的讨论请参见范例 6.5，有关 Optional.flatMap 方法的讨论请参见范例 6.4。

3.12 流的拼接

问题

用户希望将两个或多个流合并为一个流。

方案

Stream.concat 方法用于合并两个流。如果需要合并多个流，请使用 Stream.flatMap 方法。

讨论

假设我们从多个信源获取到数据，且希望使用流来处理其中的每个元素。一种方案是采用 Stream 接口定义的 concat 方法，其签名如下：

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

concat 方法将创建一个惰性的拼接流（lazily concatenated stream），其元素是第一个流的所有元素，后跟第二个流的所有元素。根据 Javadoc 的描述，如果两个输入流均为有序流，则生成的流也是有序流；如果**某个**输入流为并行流，则生成的流也是并行流。关闭生成的流也会关闭两个输入流。



两个输入流所包含的元素类型必须相同。

例 3-59 显示了拼接两个流的简单示例。

例 3-59　拼接两个流

```
@Test
public void concat() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    List<String> strings = Stream.concat(first, second) ❶
            .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c", "X", "Y", "Z");
    assertEquals(stringList, strings);
}
```

❶ 将第二个流的元素附加到第一个流的元素之后

如果需要添加第三个流，可以嵌套使用拼接操作，如例 3-60 所示。

例 3-60　拼接多个流（concat 方法）

```
@Test
public void concatThree() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");

    List<String> strings = Stream.concat(Stream.concat(first, second), third)
            .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

尽管嵌套拼接的方案可行，但请注意 Javadoc 所做的注释：

> 通过重复拼接操作构建流时应谨慎行事。访问一个深度拼接流中的元素可能导致深层调用链（deep call chain）甚至抛出 StackOverflowException。

换言之，concat 方法实际上构建了一个流的二叉树（binary tree），使用过多就会变得难以处理。

另一种方案是采用 reduce 方法执行多次拼接操作，如例 3-61 所示。

例 3-61　拼接多个流（reduce 方法）

```
@Test
public void reduce() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    List<String> strings = Stream.of(first, second, third, fourth)
            .reduce(Stream.empty(), Stream::concat)   ❶
            .collect(Collectors.toList());

    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
```

```
        assertEquals(stringList, strings);
    }
```

❶ 对空流使用 reduce 方法和二元运算符

由于用作方法引用的 concat 方法属于二元运算符，上述程序同样有效。不过请注意，虽然代码更简洁，但并不能解决潜在的栈溢出（stack overflow）问题。

有鉴于此，在合并多个流时，使用 flatMap 方法成为一种自然而然的解决方案，如例 3-62 所示。

例 3-62　拼接多个流（flatMap 方法）

```
    @Test
    public void flatMap() throws Exception {
        Stream<String> first = Stream.of("a", "b", "c").parallel();
        Stream<String> second = Stream.of("X", "Y", "Z");
        Stream<String> third = Stream.of("alpha", "beta", "gamma");
        Stream<String> fourth = Stream.empty();

        List<String> strings = Stream.of(first, second, third, fourth)
                .flatMap(Function.identity())
                .collect(Collectors.toList());
        List<String> stringList = Arrays.asList("a", "b", "c",
            "X", "Y", "Z", "alpha", "beta", "gamma");
        assertEquals(stringList, strings);
    }
```

上述代码可以运行，但仍然有其不足。如果任何一个输入流为并行流，那么通过 concat 方法创建的流也是并行流，但 flatMap 方法返回的则不是并行流（例 3-63）。

例 3-63　并行流还是非并行流

```
    @Test
    public void concatParallel() throws Exception {
        Stream<String> first = Stream.of("a", "b", "c").parallel();
        Stream<String> second = Stream.of("X", "Y", "Z");
        Stream<String> third = Stream.of("alpha", "beta", "gamma");

        Stream<String> total = Stream.concat(Stream.concat(first, second), third);

        assertTrue(total.isParallel());
    }

    @Test
    public void flatMapNotParallel() throws Exception {
        Stream<String> first = Stream.of("a", "b", "c").parallel();
        Stream<String> second = Stream.of("X", "Y", "Z");
        Stream<String> third = Stream.of("alpha", "beta", "gamma");
        Stream<String> fourth = Stream.empty();

        Stream<String> total = Stream.of(first, second, third, fourth)
                .flatMap(Function.identity());
        assertFalse(total.isParallel());
    }
```

尽管如此，只要尚未开始处理数据，总是可以通过调用 parallel 方法来实现并行流（例 3-64）。

例 3-64　将 flatMap 方法返回的流转换为并行流

```
@Test
public void flatMapParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    Stream<String> total = Stream.of(first, second, third, fourth)
            .flatMap(Function.identity());
    assertFalse(total.isParallel());

    total = total.parallel();
    assertTrue(total.isParallel());
}
```

如上所示，由于 flatMap 属于中间操作，可以通过 parallel 方法对流进行修改。

简而言之，concat 方法适用于两个流的拼接，可以作为一种一般性归约操作使用，flatMap 方法则更具普遍意义。

另见

感兴趣的话可以阅读一篇不错的博文“Efficient multiple-stream concatenation in Java”，文章描述了将多个流合并为一个流时需要考虑的性能问题。

有关 Stream.flatMap 方法的讨论请参见范例 3.11。

3.13　惰性流

问题

用户希望处理满足条件所需的最小数量的流元素。

方案

流是惰性的，在达到终止条件前不会处理元素，达到终止条件后逐个处理每个元素。如果遇到短路操作，那么只要满足所有条件，流处理就会终止。

讨论

读者在第一次接触流处理时，很容易认为流处理的效率不高。如例 3-65[7] 所示，我们将 100 到 200 之间的所有整数倍增，然后找出能被 3 整除的第一个整数。

注 7：感谢 Venkat Subramaniam 博士为本例所做的贡献。

例 3-65　将 100 到 200 之间的所有整数倍增，再找出能被 3 整除的第一个整数

```
OptionalInt firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(n -> n * 2)
    .filter(n -> n % 3 == 0)
    .findFirst();
System.out.println(firstEvenDoubleDivBy3); ❶
```

❶ 打印 `Optional[204]`

如果不了解流处理的机制，读者可能认为上述代码做了不少无用功：

- 创建 100 到 199 之间的整数（100 次操作）
- 将每个整数倍增（100 次操作）
- 校验每个整数能否被 3 整除（100 次操作）
- 返回结果流的第一个元素（1 次操作）

那么，既然满足要求的第一个值为 204，为什么还要处理所有其他的数字呢？

不要误会，流处理的机制并非如此。流是**惰性**的，在达到终止条件前不会处理元素，达到终止条件后才通过流水线逐一处理每个元素。为说明这一点，例 3-66 对代码进行重构，以便读者观察每个元素通过流水线时的情况。

例 3-66　对每个流元素进行显式处理

```
public int multByTwo(int n) {      ❶
    System.out.printf("Inside multByTwo with arg %d%n", n);
    return n * 2;
}

public boolean divByThree(int n) { ❷
    System.out.printf("Inside divByThree with arg %d%n", n);
    return n % 3 == 0;
}

// 其他代码

firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(this::multByTwo)          ❶
    .filter(this::divByThree)      ❷
    .findFirst();
```

❶ 用于倍增（并打印）的方法引用

❷ 用于对 3 取模（并打印）的方法引用

例 3-66 的输出如下：

```
Inside multByTwo with arg 100
Inside divByThree with arg 200
Inside multByTwo with arg 101
Inside divByThree with arg 202
Inside multByTwo with arg 102
Inside divByThree with arg 204
First even divisible by 3 is Optional[204]
```

在本例中，100 被映射到 200，它未通过筛选器，流移至 101；101 被映射到 202，它同样未通过筛选器；下一个值 102 被映射到 204，它能被 3 整除，因此通过筛选器。流处理在**仅处理三个值后**即告终止，一共进行了 6 次操作。

这是流处理相对于直接处理集合的最大优点之一。对集合而言，必须执行完所有操作才能进行下一步操作。对流而言，各种中间操作构成了一条流水线，但流在达到终止操作前不会处理任何元素，达到终止操作后只处理所需的值。

流处理并非在任何情况下都有意义：如果进行任何状态操作（如排序或求和），就不得不处理所有值。但是，如果无状态操作后跟一个短路终止操作，流处理的优点还是很明显的。

另见

有关 `findFirst` 和 `findAny` 方法之间的差异请参见范例 3.9。

第 4 章

比较器与收集器

Java 8 为 `java.util.Comparator` 接口新增了多种静态和默认方法，使排序操作变得更为简单。现在，只需通过一系列库调用，就能根据一个属性对 POJO 集合进行排序。

Java 8 还引入了一个新的工具类 `java.util.stream.Collectors`，它提供将流转换回各类集合所需的静态方法。此外，收集器也可以在“下游”使用，利用它们对分组（grouping）或分区（partitioning）操作进行后期处理。

这一章的范例将讨论上述各种概念。

4.1 利用比较器实现排序

问题

用户希望实现对象的排序。

方案

使用传入 `Comparator` 的 `Stream.sorted` 方法，`Comparator` 既可以通过 lambda 表达式实现，也可以使用 `Comparator` 接口定义的某种 `comparing` 方法生成。

讨论

`Stream.sorted` 方法根据类的自然顺序（natural ordering）生成一个新的排序流，自然顺序是通过实现 `java.util.Comparable` 接口来指定的。

如例 4-1 所示，我们对字符串集合进行排序。

例 4-1　根据字典序（自然顺序）对字符串排序

```
private List<String> sampleStrings =
    Arrays.asList("this", "is", "a", "list", "of", "strings");

public List<String> defaultSort() {
    Collections.sort(sampleStrings);  ❶
    return sampleStrings;
}

public List<String> defaultSortUsingStreams() {
    return sampleStrings.stream()
        .sorted()                     ❷
        .collect(Collectors.toList());
}
```

❶ 默认排序（Java 7 及之前的版本）

❷ 默认排序（Java 8 及之后的版本）

从 Java 1.2 引入集合框架（collections framework）开始，工具类 `Collections` 就已存在。`Collections` 类定义的静态方法 `sort` 传入 `List` 作为参数，但返回 `void`。这种排序是破坏性的，会修改所提供的集合。换言之，`Collections.sort` 方法不符合 Java 8 所倡导的将不可变性（immutability）置于首要位置的函数式编程原则。

Java 8 采用 `Stream.sorted` 方法实现相同的排序，但不对原始集合进行修改，而是生成一个新的流。在例 8-1 中，完成集合的排序后，程序根据类的自然顺序对返回列表进行排序。对于字符串，自然顺序是字典序（lexicographic order）；如果所有字符串均为小写，自然顺序就相当于字母顺序（alphabetical order），从本例可以观察到这一点。

如果希望以其他方式排序字符串，可以使用 `sorted` 方法的重载形式，传入 `Comparator` 作为参数。

例 4-2 采用两种不同的方式，根据长度对字符串进行排序。

例 4-2　根据长度对字符串排序

```
public List<String> lengthSortUsingSorted() {
    return sampleStrings.stream()
        .sorted((s1, s2) -> s1.length() - s2.length()) ❶
        .collect(toList());
}

public List<String> lengthSortUsingComparator() {
return sampleStrings.stream()
    .sorted(Comparator.comparingInt(String::length))   ❷
    .collect(toList());
}
```

❶ 使用 lambda 表达式作为 `Comparator`，根据长度进行排序

❷ 使用 `Comparator.comparingInt` 方法

`sorted` 方法的参数为 `java.util.Comparator`，它是一种函数式接口。对于第一个方法 `lengthSortUsingSorted`，所提供的 lambda 表达式用于实现 `Comparator.compare` 方法。在

Java 7 及之前的版本中，实现通常由匿名内部类提供，而本例仅需要一个 lambda 表达式。



Java 8 引入 `sort(Comparator)` 作为 `List` 接口的默认实例方法，它相当于 `Collections` 类的 `sort(List, Comparator)` 方法。由于两种方法返回 `void` 且都是破坏性的，本节讨论的 `Stream.sorted(Comparator)` 方法（返回一个新的排序流）仍然是首选方案。

第二个方法 `lengthSortUsingComparator` 利用了 `Comparator` 接口新增的某种静态方法。`comparingInt` 方法传入一个 `ToIntFunction` 类型的参数（文档称之为 `keyExtractor`），用于将字符串转换为整型数据，并生成一个使用该 `keyExtractor` 对集合进行排序的 `Comparator`。

`Comparator` 接口新增的各种默认方法极为有用。虽然不难写出一个根据长度进行排序的 `Comparator`，但如果需要对多个字段排序，代码可能会变得很复杂。例如，我们希望根据长度对字符串排序，如果长度相同则按字母顺序排序。如例 4-3 所示，采用 `Comparator` 接口提供的默认和静态方法很容易就能解决这个问题。

例 4-3　根据长度对字符串排序，长度相同则按字母顺序排序

```
public List<String> lengthSortThenAlphaSort() {
    return sampleStrings.stream()
        .sorted(comparing(String::length)            ❶
                    .thenComparing(naturalOrder()))
        .collect(toList());
}
```

❶ 根据长度对字符串排序，长度相同则按字母顺序排序

`Comparator` 接口定义了一个称为 `thenComparing` 的默认方法。与 `comparing` 方法类似，`thenComparing` 方法也传入 `Function` 作为参数，文档同样将其称为 `keyExtractor`。如果将 `thenComparing` 方法链接到 `comparing` 方法会返回 `Comparator`，它首先比较第一个数量，如果相同则比较第二个数量，以此类推。

静态导入通常使代码更易于阅读。一旦熟悉 `Comparator` 接口和 `Collectors` 类定义的各种静态方法，就能写出更简单的代码。在本例中，`comparing` 和 `naturalOrder` 方法已被静态导入。

即便没有实现 `Comparable` 接口，上述方案也适用于任何类。如例 4-4 所示，我们定义一个描述高尔夫球手的 `Golfer` 类。

例 4-4　描述高尔夫球手的 `Golfer` 类

```
public class Golfer {
    private String first;
    private String last;
    private int score;

    // 其他方法
}
```

为了创建一个高尔夫锦标赛排行榜，可以依次根据各个球手的得分、姓氏、名字进行排序。例 4-5 显示了如何实现高尔夫球手的排序。

例 4-5　对高尔夫球手排序

```
private List<Golfer> golfers = Arrays.asList(
    new Golfer("Jack", "Nicklaus", 68),
    new Golfer("Tiger", "Woods", 70),
    new Golfer("Tom", "Watson", 70),
    new Golfer("Ty", "Webb", 68),
    new Golfer("Bubba", "Watson", 70)
);

public List<Golfer> sortByScoreThenLastThenFirst() {
    return golfers.stream()
        .sorted(comparingInt(Golfer::getScore)
                    .thenComparing(Golfer::getLast)
                    .thenComparing(Golfer::getFirst))
        .collect(toList());
}
```

调用 `sortByScoreThenLastThenFirst` 方法，输出如例 4-6 所示。

例 4-6　对高尔夫球手排序后的结果

```
Golfer{first='Jack', last='Nicklaus', score=68}
Golfer{first='Ty', last='Webb', score=68}
Golfer{first='Bubba', last='Watson', score=70}
Golfer{first='Tom', last='Watson', score=70}
Golfer{first='Tiger', last='Woods', score=70}
```

本例首先根据得分对各个球手进行排序，因此 Jack Nicklaus 和 Ty Webb[1] 排在 Tiger Woods、Bubba Watson 与 Tom Watson 之前。得分相同时根据姓氏进行排序，因此 Jack Nicklaus 排在 Ty Webb 之前，Bubba Watson 和 Tom Watson 排在 Tiger Woods 之前。如果得分和姓氏都相同，则根据名字进行排序，因此 Bubba Watson 排在 Tom Watson 之前。

借由 `Comparator` 接口定义的默认和静态方法以及 `Stream` 接口新增的 `sorted` 方法，生成复杂的排序变得易如反掌。

4.2　将流转换为集合

问题

流处理完成之后，用户希望将流转换为 `List`、`Set` 或其他线性集合。

方案

使用 `Collectors` 工具类定义的 `toList`、`toSet` 或 `toCollection` 方法。

注 1：当然，Ty Webb 是电影《小小球童》中的角色。Judge Smails 问道："今天打得怎么样？" Ty Webb 回答："我没有计分。" Smails 继续问道："那你怎么衡量自己与其他球手的水平呢？" Webb 回答："身高。" 根据身高进行排序留给读者作为简单的练习。

讨论

Java 8 一般通过称为流水线（pipeline）的中间操作（intermediate operation）来传递流元素，并在达到终止操作（terminal operation）后结束。Stream 接口定义的 collect 方法就是一种终止操作，用于将流转换为集合。

collect 方法有两种重载形式，如例 4-7 所示。

例 4-7　Stream.collect 方法

```
<R,A> R collect(Collector<? super T,A,R> collector)
<R>   R collect(Supplier<R> supplier,
                BiConsumer<R,? super T> accumulator,
                BiConsumer<R,R> combiner)
```

本范例采用第一种形式，它传入 Collector 作为参数。收集器执行“可变归约操作”（mutable reduction operation），将元素累加至结果容器（result container），此时结果是一个集合。

由于 java.util.stream.Collector 属于接口，无法被实例化。Collector 接口包含一个称为 of 的静态方法，它用于生成 Collector，但通常有更好（或至少更简单）的方式可以实现这一点。



Java 8 API 经常使用静态方法 of 作为工厂方法。

Collectors 类定义的静态方法将用于生成 Collector 实例，它作为 Stream.collect 方法的参数来填充集合。

例 4-8[2] 显示了一个创建 List 的简单示例。

例 4-8　创建 List

```
List<String> superHeroes =
    Stream.of("Mr. Furious", "The Blue Raja", "The Shoveler",
              "The Bowler", "Invisible Boy", "The Spleen", "The Sphinx")
          .collect(Collectors.toList());
```

上述方法创建了一个 ArrayList 类，并采用给定的流元素进行填充。创建 Set 也很简单，如例 4-9 所示。

例 4-9　创建 Set

```
Set<String> villains =
    Stream.of("Casanova Frankenstein", "The Disco Boys",
              "The Not-So-Goodie Mob", "The Suits", "The Suzies",
```

注 2：本范例中出现的人名来自《神秘兵团》，这是 20 世纪 90 年代最为人所忽视的电影之一。（Furious 说：“Lance Hunt 就是神奇队长。”Shoveler 答道：“Lance Hunt 戴眼镜，但神奇队长不戴。”Furious 说：“他变身时会把眼镜摘下来。”Shoveler 答道：“怎么可能！他根本看不见！”）

```
                "The Furriers", "The Furriers")  ❶
            .collect(Collectors.toSet());
}
```

❶ 重复的人名（将在转换为 Set 时删除）

上述方法创建了一个 HashSet 类的实例并加以填充，并忽略任何重复的人名。

例 4-8 和例 4-9 均使用默认的数据结构：对于 List 是 ArrayList，对于 Set 是 HashSet。如果希望指定某种特定的数据结构，则应使用 Collectors.toCollection 方法，它传入 Supplier 作为参数。示例代码如例 4-10 所示。

例 4-10　创建关联列表（linked list）

```
List<String> actors =
    Stream.of("Hank Azaria", "Janeane Garofalo", "William H. Macy",
              "Paul Reubens", "Ben Stiller", "Kel Mitchell", "Wes Studi")
          .collect(Collectors.toCollection(LinkedList::new));
}
```

由于 toCollection 方法的参数是集合 Supplier，本例提供 LinkedList 类的构造函数引用。collect 方法将 LinkedList 实例化，并采用给定的姓名加以填充。

Stream 接口还定义了一个用于创建对象数组的方法 toArray，它有两种重载形式：

```
    Object[] toArray();
<A> A[]      toArray(IntFunction<A[]> generator);
```

第一种形式返回一个包含流元素的数组，但未指定类型。第二种形式传入一个函数并生成所需类型的新数组，数组的长度与流相同，很容易与数组构造函数引用（array constructor reference）一起使用，如例 4-11 所示。

例 4-11　创建 Array

```
String[] wannabes =
    Stream.of("The Waffler", "Reverse Psychologist", "PMS Avenger")
          .toArray(String[]::new); ❶
}
```

❶ 数组构造函数引用作为 Supplier

返回数组具有指定的类型，其长度与流中的元素数量匹配。

为了将流转换为 Map，Collectors.toMap 方法需要传入两个 Function 实例，分别用于键和值。

我们以一个包装 name 和 role 的 Actor POJO 为例进行讨论。假设一部给定电影中存在 Actor 实例的 Set，例 4-12 根据这些实例创建了一个 Map。

例 4-12　创建 Map

```
Set<Actor> actors = mysteryMen.getActors();

Map<String, String> actorMap = actors.stream()
    .collect(Collectors.toMap(Actor::getName, Actor::getRole));  ❶

actorMap.forEach((key,value) ->
    System.out.printf("%s played %s%n", key, value));
```

❶ 生成键和值所用的函数

输出如下：

```
Janeane Garofalo played The Bowler
Greg Kinnear played Captain Amazing
William H. Macy played The Shoveler
Paul Reubens played The Spleen
Wes Studi played The Sphinx
Kel Mitchell played Invisible Boy
Geoffrey Rush played Casanova Frankenstein
Ben Stiller played Mr. Furious
Hank Azaria played The Blue Raja
```

利用 `Collectors.toConcurrentMap` 方法对本例稍作修改，就能创建 `ConcurrentMap`。

另见

有关 `Supplier` 接口的讨论请参见范例 2.2，有关构造函数引用的讨论请参见范例 1.3，有关 `toMap` 方法的讨论请参见范例 4.3。

4.3 将线性集合添加到映射

问题

用户希望将对象集合添加到 `Map`，其中键为某种对象属性，值为对象本身。

方案

使用 `Collectors` 类定义的 `toMap` 方法以及 `Function` 接口定义的 `identity` 方法。

讨论

这是一个简短且非常集中的用例，但本节讨论的解决方案可能为实际开发提供很大便利。

假设存在一个由 `Book` 实例构成的 `List`。`Book` 是一个简单的 POJO，由 ID、书名、价格参数构成。`Book` 类的简单形式如例 4-13 所示。

例 4-13　Book 类（描述图书的简单 POJO）

```
public class Book {
    private int id;
    private String name;
    private double price;

    // 其他方法
}
```

此外，假设存在一个由 `Book` 实例构成的集合，如例 4-14 所示。

例 4-14　图书集合

```
List<Book> books = Arrays.asList(
    new Book(1, "Modern Java Recipes", 49.99),
    new Book(2, "Java 8 in Action", 49.99),
    new Book(3, "Java SE8 for the Really Impatient", 39.99),
    new Book(4, "Functional Programming in Java", 27.64),
    new Book(5, "Making Java Groovy", 45.99)
    new Book(6, "Gradle Recipes for Android", 23.76)
);
```

很多情况下，我们需要的可能是 `Map` 而非 `List`，`Map` 的键为图书 ID，值为图书本身。借由 `Collectors.toMap` 方法，很容易就能将图书添加到 `Map`，例 4-15 显示了两种不同的方案。

例 4-15　将图书添加到 `Map`

```
Map<Integer, Book> bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, b -> b));              ❶

bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, Function.identity()));  ❷
```

❶ lambda 标识：给定一个元素并返回

❷ 静态方法 `Function.identity` 可以实现同样的目的

`toMap` 方法传入两个 `Function` 实例作为参数，根据所提供的对象，两个函数分别生成键和值。在本例中，键由 `Book::getId` 映射，值为图书本身。

可以看到，第一个 `toMap` 方法传入两个参数，一个是映射到键的 `getId`，另一个是返回参数的显式 lambda 表达式。第二个 `toMap` 方法通过静态方法 `Function.identity` 实现相同的目的。

两种静态 identity 方法

静态方法 `Function.identity` 的签名如下：

```
static <T> Function<T,T>      identity()
```

它在 Java 标准库中的实现如例 4-16 所示。

例 4-16　`Function.identity` 方法

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

`UnaryOperator` 接口是 `Function` 的子接口，但无法重写静态方法。根据 Javadoc 的描述，`UnaryOperator` 接口也声明了一个称为 `identity` 的静态方法：

```
static <T> UnaryOperator<T>     identity()
```

`UnaryOperator.identity` 方法在 Java 标准库中的实现与 `Function.identity` 方法基本相同，如例 4-17 所示。

例 4-17　UnaryOperator.identity 方法

```
static <T> UnaryOperator<T> identity() {
    return t -> t;
}
```

二者的区别仅在于调用方式（不同的接口名）和相应的返回类型有所不同。使用哪种 identity 方法均可，这里将两种方法列出供读者参考。

无论是提供显式的 lambda 表达式抑或使用静态方法，只是编程风格不同而已。两种方案都能很容易地将集合值添加到 Map，其中键为对象的属性，值为对象本身。

另见

有关 Function 接口、一元运算符与二元运算符的讨论请参见范例 2.4。

4.4　对映射排序

问题

用户希望根据键或值对 Map 排序。

方案

使用 Map.Entry 接口新增的静态方法。

讨论

Map 接口始终包含一个称为 Map.Entry 的公共静态内部接口（public, static, inner interface），它表示一个键值对。Map 接口定义的 entrySet 方法返回 Map.Entry 元素的 Set。在 Java 8 之前，getKey 和 getValue 是 Map.Entry 接口两种最常用的方法，二者分别返回与某个条目对应的键和值。

Java 8 为 Map.Entry 接口引入了一些新的静态方法，如表 4-1 所示。

表4-1　Map.Entry接口新增的静态方法（参见Java 8文档）

方法	描述
comparingByKey()	返回一个比较器，它根据键的自然顺序比较 Map.Entry
comparingByKey(Comparator<? super K> cmp)	返回一个比较器，它使用给定的 Comparator 并根据键比较 Map.Entry
comparingByValue()	返回一个比较器，它根据值的自然顺序比较 Map.Entry
comparingByValue(Comparator<? super V> cmp)	返回一个比较器，它使用给定的 Comparator 并根据值比较 Map.Entry

我们以创建单词长度与单词数量的 `Map` 为例，演示上述方法的用法（例 4-18）。所有 Unix 系统的 usr/share/dict/words 目录中都包含一个文件，它收录了《韦氏词典（第 2 版）》的内容，每个单词在文件中占据一行。`Files.lines` 方法可用于读取文件并生成一个包含这些行的字符串流。此时，流包含词典中的所有单词。

例 4-18　将词典文件读入 `Map`

```
System.out.println("\nNumber of words of each length:");
try (Stream<String> lines = Files.lines(dictionary)) {
    lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()))
        .forEach((len, num) -> System.out.printf("%d: %d%n", len, num));
} catch (IOException e) {
    e.printStackTrace();
}
```

本例的详细讨论请参见范例 7.1，目前可以这样理解。

- 文件在 `try-with-resources` 代码块内读取。由于 `Stream` 接口实现了 `AutoCloseable`，`try` 代码块执行完毕后，Java 对 `Stream` 调用 `close` 方法，然后对 `File` 调用 `close` 方法。
- 筛选器只筛出长度至少为 20 个字符的单词，以供进一步处理。
- `Collectors.groupingBy` 方法传入 `Function` 作为第一个参数，表示分类器（classifier）。在本例中，分类器是每个字符串的长度。如果 `groupingBy` 方法只传入一个参数，则结果为 `Map`，其中键为分类器的值，值为匹配分类器的元素列表。这种情况下，`groupingBy(String::length)` 将返回 `Map<Integer,List<String>>`，其中键为单词长度，值为该长度的单词列表。
- 在本例中，双参数形式的 `groupingBy` 方法传入另一个 `Collector`，它称为**下游收集器**（downstream collector），用于对单词列表进行后期处理。这种情况下，`groupingBy` 方法的返回类型是 `Map<Integer,Long>`，其中键为单词长度，值为词典中该长度的单词数量。

例 4-18 的输出结果如下：

```
Number of words of each length:
21: 82
22: 41
23: 17
24: 5
```

换言之，有 82 个单词的长度为 21，41 个单词的长度为 22，17 个单词的长度为 23，5 个单词的长度为 24[3]。

不难看到，程序按单词长度的升序打印映射中的单词。如果希望按降序打印，可以使用 `Map.Entry` 接口定义的 `comparingByKey` 方法，如例 4-19 所示。

注 3：根据记录，这 5 个最长的单词为 formaldehydesulphoxylate（甲醛次硫酸氢钠）、pathologicopsychological（病理心理学）、scientificophilosophical（科学哲学）、tetraiodophenolphthalein（四碘酚酞）以及 thyroparathyroidectomize（甲状腺甲状腺切除术）。希望拼写检查工具可以识别这些单词。

例 4-19 根据键对映射排序

```
System.out.println("\nNumber of words of each length (desc order):");
try (Stream<String> lines = Files.lines(dictionary)) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()));

    map.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
        .forEach(e -> System.out.printf("Length %d: %2d words%n",
            e.getKey(), e.getValue()));
} catch (IOException e) {
    e.printStackTrace();
}
```

返回 `Map<Integer,Long>` 之后，程序将提取 `entrySet` 并产生一个流。`Stream.sorted` 方法使用提供的比较器生成经过排序的流。

在本例中，`comparingByKey` 方法返回一个根据键进行排序的比较器。如果希望以键的相反顺序排序，可以使用 `comparingByKey` 方法的重载形式，它传入比较器作为参数。



`Stream.sorted` 方法生成一个新的排序流，它不对源数据进行修改。换言之，原始 `Map` 不受影响。

例 4-19 的输出结果如下：

```
Number of words of each length (desc order):
Length 24:  5 words
Length 23: 17 words
Length 22: 41 words
Length 21: 82 words
```

表 4-1 列出的 `comparingByValue` 方法，用法与 `comparingByKey` 类似。

另见

有关根据键或值对 `Map` 进行排序的其他示例请参见附录 A，有关下游收集器的讨论请参见范例 4.6，有关词典中的文件处理请参见范例 7.1。

4.5 分区与分组

问题

用户希望将元素集合分为若干个类别。

方案

Collectors.partitioningBy 方法将元素拆分为满足 Predicate 与不满足 Predicate 的两类。Collectors.groupingBy 方法生成一个由类别构成的 Map，其中值为每个类别中的元素。

讨论

假设存在一个由字符串构成的集合，可以通过 partitioningBy 方法将这些字符串按偶数长度和奇数长度进行划分，如例 4-20 所示。

例 4-20　根据偶数或奇数长度对字符串分区

```
List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
        "strings", "to", "use", "as", "a", "demo");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0)); ❶

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n", key, value));
//
// false: [a, strings, use, a]
// true: [this, is, long, list, of, to, as, demo]
```

❶ 根据偶数或奇数长度进行分区

partitioningBy 方法包括两种形式：

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
    Predicate<? super T> predicate)
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

可以看到，返回类型中涉及泛型（generics），因此略显复杂，不过实际开发中很少会用到它们。两种 partitioningBy 方法的结果成为 collect 方法的参数，该方法使用生成的收集器来创建由第三个泛型参数定义的输出映射。

第一种 partitioningBy 方法传入单个 Predicate 作为参数，它将元素分为满足 Predicate 与不满足 Predicate 的两类。我们总是可以得到一个恰好包含两个条目的 Map，其中一个值列表满足 Predicate，另一个则不满足 Predicate。

partitioningBy 方法的重载形式传入 Collector 作为第二个参数，它称为**下游收集器**。它支持对分区返回的列表进行后期处理，相关讨论请参见范例 4.6。

而 groupingBy 方法执行的操作类似于 SQL 的 GROUP BY 语句。该方法返回一个 Map，其中键为分组，值为各个分组中的元素列表。



如果从数据库获取数据，请务必将分组操作放在数据库中执行，因为 Java 8 新增的方法适合处理内存中的数据。

groupingBy 方法的签名如下：

```
static <T,K> Collector<T,?,Map<K,List<T>>>      groupingBy(
    Function<? super T,? extends K> classifier)
```

Function 参数传入流的各个元素，并提取需要分组的元素。接下来，我们不是将字符串简单地分为两类，而是根据长度进行划分，如例 4-21 所示。

例 4-21　根据长度对字符串分组

```
List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
        "strings", "to", "use", "as", "a", "demo");

Map<Integer, List<String>> lengthMap = strings.stream()
    .collect(Collectors.groupingBy(String::length)); ❶

lengthMap.forEach((k,v) -> System.out.printf("%d: %s%n", k, v));
//
// 1: [a, a]
// 2: [is, of, to, as]
// 3: [use]
// 4: [this, long, list, demo]
// 7: [strings]
```

❶ 根据长度进行分组

对于所生成的映射，键为字符串长度（1、2、3、4、7），值为各个长度的字符串列表。

另见

作为对本范例的延伸，范例 4.6 将讨论如何对 groupingBy 或 partitioningBy 操作返回的列表进行后期处理。

4.6　下游收集器

问题

用户希望对 groupingBy 或 partitioningBy 操作返回的集合进行后期处理。

方案

使用 java.util.stream.Collectors 类定义的某种静态工具方法。

讨论

有关将元素划分为多个类别的讨论请参见范例 4.5。groupingBy 和 partitioningBy 方法返回的是 Map，其中键为类别（对于 partitioningBy 方法是布尔值 true 或 false，对于 groupingBy 方法是对象），值为满足各个类别的元素列表。读者或许还记得根据偶数和奇数长度对字符串进行分区的示例（例 4-20），为便于参考，例 4-22 完整复制了这个示例。

例 4-22　根据偶数或奇数长度对字符串分区

```
List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
        "strings", "to", "use", "as", "a", "demo");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0));

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n", key, value));
//
// false: [a, strings, use, a]
// true: [this, is, long, list, of, to, as, demo]
```

较之实际的列表，我们或许对每个类别包含多少元素更感兴趣。换言之，我们可能只需要各个列表中的元素数量，而不是返回 `Map`（值为 `List<String>`）。`partitioningBy` 方法的重载形式如下，其第二个参数为 `Collector` 类型：

```
static <T,D,A> Collector<T,?,Map<Boolean,D>>     partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

静态方法 `Collectors.counting` 的作用就在于此，其用法如例 4-23 所示。

例 4-23　对已分区的字符串进行计数

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0,
                Collectors.counting()));  ❶

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n", k, v));
//
// false: 4
// true: 8
```

❶ 下游收集器

这就是所谓的**下游收集器**，它对下游的结果列表（即在分区操作完成之后）进行后期处理。

`groupingBy` 方法也有一种传入下游收集器的重载形式：

```
/**
* @param <T>：输入元素的类型
* @param <K>：键的类型
* @param <A>：下游收集器的中间累加类型
* @param <D>：下游归约操作的结果类型
* @param classifier：将输入元素映射到键的分类器函数
* @param downstream：实现下游归约操作的Collector
* @return：实现级联分组操作的Collector
*/
static <T,K,A,D> Collector<T,?,Map<K,D>>        groupingBy(
    Function<? super T,? extends K> classifier,
    Collector<? super T,A,D> downstream)
```

方法签名中包含了部分 Javadoc 注释，其中 `T` 为集合中元素的类型，`K` 为结果映射的键类型，`A` 为累加器，`D` 为下游收集器的类型，`?` 表示“未知”。有关泛型在 Java 8 中的应用，详细信息请参见附录 A。

Stream 接口定义的部分方法在 Collectors 类中存在类似的对应，它们的对比如表 4-2 所示。

表4-2：Stream接口定义的方法与Collectors类定义的方法

Stream	Collectors
count	counting
map	mapping
min	minBy
max	maxBy
IntStream.sum	summingInt
DoubleStream.sum	summingDouble
LongStream.sum	summingLong
IntStream.summarizing	summarizingInt
DoubleStream.summarizing	summarizingDouble
LongStream.summarizing	summarizingLong

需要再次强调的是，下游收集器用于对上游操作（如分区或分组）产生的对象集合进行后期处理。

另见

有关应用下游收集器确定词典中最长单词的讨论请参见范例 7.1，有关 partitioningBy 和 groupingBy 方法的深入讨论请参见范例 4.5，有关泛型的详细信息请参见附录 A。

4.7 查找最大值和最小值

问题

用户希望确定流中的最大值或最小值。

方案

既可以使用 BinaryOperator 接口定义的 maxBy 和 minBy 方法，也可以使用 Stream 接口定义的 max 和 min 方法，还可以使用 Collectors 类定义的 maxBy 和 minBy 方法。

讨论

BinaryOperator 是 java.util.function 包定义的一种函数式接口，它继承自 BiFunction 接口，适合在函数和返回值的参数属于同一个类时使用。

BinaryOperator 接口包括两种静态方法：

```
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

两种方法根据所提供的 Comparator，返回一个 BinaryOperator。

我们以一个 Employee POJO 为例，讨论如何获取流的最大值。如例 4-24 所示，Employee POJO 包括 name、salary 与 department 这三个特性。

例 4-24　Employee POJO

```
public class Employee {
    private String name;
    private Integer salary;
    private String department;

    // 其他方法
}

List<Employee> employees = Arrays.asList(                        ❶
        new Employee("Cersei",     250_000, "Lannister"),
        new Employee("Jamie",      150_000, "Lannister"),
        new Employee("Tyrion",       1_000, "Lannister"),
        new Employee("Tywin",    1_000_000, "Lannister"),
        new Employee("Jon Snow",    75_000, "Stark"),
        new Employee("Robb",       120_000, "Stark"),
        new Employee("Eddard",     125_000, "Stark"),
        new Employee("Sansa",            0, "Stark"),
        new Employee("Arya",         1_000, "Stark"));

Employee defaultEmployee =                                       ❷
    new Employee("A man (or woman) has no name", 0, "Black and White");
```

❶ 员工集合

❷ 流为空时的默认值

给定一个由员工构成的集合，可以使用 Stream.reduce 方法，传入 BinaryOperator 作为参数。例 4-25 展示了如何查找工资最高的员工信息。

例 4-25　BinaryOperator.maxBy 方法的应用

```
Optional<Employee> optionalEmp = employees.stream()
    .reduce(BinaryOperator.maxBy(Comparator.comparingInt(Employee::getSalary)));

System.out.println("Emp with max salary: " +
    optionalEmp.orElse(defaultEmployee));
```

请注意，reduce 方法需要传入 BinaryOperator 作为参数。静态方法 maxBy 根据所提供的 Comparator 生成该 BinaryOperator，并按工资高低对员工进行比较。

上述方案是可行的，不过采用 Stream.max 方法其实更简单，该方法可以直接应用于流：

```
Optional<T> max(Comparator<? super T> comparator)
```

例 4-26 显示了 max 方法的应用。

例 4-26　Stream.max 方法的应用

```
optionalEmp = employees.stream()
        .max(Comparator.comparingInt(Employee::getSalary));
```

Stream.max 方法与 BinaryOperator.maxBy 方法的结果并无不同。

此外，几种基本类型流（IntStream、LongStream 与 DoubleStream）也提供一个不传入任何参数的 max 方法，其应用如例 4-27 所示。

例 4-27　查找最高工资

```
OptionalInt maxSalary = employees.stream()
        .mapToInt(Employee::getSalary)
        .max();
System.out.println("The max salary is " + maxSalary);
```

在本例中，mapToInt 方法通过调用 getSalary 方法将员工流转换为整数流，并返回 IntStream。之后，Max 方法返回 OptionalInt。

类似地，Collectors 工具类也定义了一种称为 maxBy 的静态方法，可以直接用于查找最高工资，如例 4-28 所示。

例 4-28　Collectors.maxBy 方法的应用

```
optionalEmp = employees.stream()
    .collect(Collectors.maxBy(Comparator.comparingInt(Employee::getSalary)));
```

但是，Collectors.maxBy 方法不便处理，最好采用 Stream.max 方法作为替代（如例 4-27 所示）。Collectors.maxBy 方法在用作下游收集器（即对分组或分区操作进行后期处理）时相当有用。例 4-29 通过 Collectors.groupingBy 方法创建了一个部门到员工列表的映射，然后计算每个部门中工资最高的员工。

例 4-29　Collectors.maxBy 用作下游收集器

```
Map<String, Optional<Employee>> map = employees.stream()
    .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.maxBy(
                    Comparator.comparingInt(Employee::getSalary))));

map.forEach((house, emp) ->
        System.out.println(house + ": " + emp.orElse(defaultEmployee)));
```

BinaryOperator.minBy 和 Collectors.minBy 方法的用法与相应的 maxBy 方法类似。

另见

有关 Function 接口的讨论请参见范例 2.4，有关下游收集器的讨论请参见范例 4.6。

4.8　创建不可变集合

问题

用户希望利用 Stream API 创建不可变的列表、集合或映射。

方案

使用 Collectors 类新增的静态方法 collectingAndThen。

讨论

函数式编程强调并行（parallelization）以及语义的清晰，倾向于尽可能使用不可变对象。Java 1.2 引入了集合框架，提供以现有集合为基础创建不可变集合的各种方法，但使用起来有些不便。

`Collections` 工具类定义了 `unmodifiableList`、`unmodifiableSet` 与 `unmodifiableMap` 方法（以及其他以 `unmodifiable` 为前缀的方法），如例 4-30 所示。

例 4-30　`Collections` 类定义的以 `unmodifiable` 为前缀的方法

```
static <T> List<T>    unmodifiableList(List<? extends T> list)
static <T> Set<T>     unmodifiableSet(Set<? extends T> s)
static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
```

上述三种方法的参数分别为现有的列表、集合与映射。结果列表、集合与映射中包含的元素和参数相同，但存在一个重要的区别：所有可以修改集合的方法（如 `add` 或 `remove`）现在都能抛出 `UnsupportedOperationException`。

在 Java 8 之前，如果通过可变参数列表获取到单个值作为参数，则会生成一个不可修改的列表或集合，如例 4-31 所示。

例 4-31　创建不可修改的列表或集合（Java 8 之前）

```
@SafeVarargs   ❶
public final <T> List<T> createImmutableListJava7(T... elements) {
    return Collections.unmodifiableList(Arrays.asList(elements));
}

@SafeVarargs   ❶
public final <T> Set<T> createImmutableSetJava7(T... elements) {
    return Collections.unmodifiableSet(new HashSet<>(Arrays.asList(elements)));
}
```

❶ 用户承诺不会破坏输入数组类型。详见附录 A。

两段代码首先传入输入值，并将它们转换为 `List`。第一段代码使用 `unmodifiableList` 包装生成的列表。而对于 `Set`，第二段代码先将列表用作集合构造函数的参数，再使用 `unmodifiableSet`。

借由 Java 8 引入的 Stream API，我们可以使用 `Collectors` 类定义的静态方法 `collectingAndThen`，如例 4-32 所示。

例 4-32　创建不可修改的列表或集合（Java 8）

```
import static java.util.stream.Collectors.collectingAndThen;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

// 采用以下方法来定义类

@SafeVarargs
public final <T> List<T> createImmutableList(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toList(),
```

```java
                    Collections::unmodifiableList)); ❶
}

@SafeVarargs
public final <T> Set<T> createImmutableSet(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toSet(),
                    Collections::unmodifiableSet)); ❶
}
```

❶ "终止器"对生成的集合进行包装

`collectingAndThen` 方法传入两个参数，一个是下游 `Collector`，另一个是称为**终止器**（finisher）的 `Function`。该方法的作用是读取输入元素，并将它们收集到 `List` 或 `Set`，然后利用不可修改的函数包装结果集合。

将一系列输入元素转换为一个不可修改的 `Map` 看起来并不直观，部分原因在于很难看出哪些输入元素将作为键，哪些将作为值。例 4-33[4] 采用实例初始化器（instance initializer），以一种很别扭的方式创建了一个不可变的 `Map`。

例 4-33　创建不可变的 `Map`

```java
Map<String, Integer> map = Collections.unmodifiableMap(
  new HashMap<String, Integer>() {{
    put("have", 1);
    put("the", 2);
    put("high", 3);
    put("ground", 4);
}});
```

熟悉 Java 9 的读者想必已经了解，本范例中的所有问题都可以通过工厂方法 `List.of`、`Set.of` 与 `Map.of` 来解决，这些方法能极大提高效率。

另见

Java 9 新增的工厂方法能自动创建不可变集合，相关讨论请参见范例 10.3。

4.9　实现Collector接口

问题

由于 `java.util.stream.Collectors` 类提供的工厂方法无法满足需要，用户希望手动实现 `java.util.stream.Collector` 接口。

方案

为工厂方法 `Collector.of` 传入的 `supplier`、`accumulator`、`combiner`、`finisher` 函数提供 lambda 表达式或方法引用，以及其他所需的特性。

注 4：灵感源自 Carl Martensen 的博文"Java 9's Immutable Collections Are Easier To Create But Use With Caution"。

讨论

Collectors 工具类定义了多种便利的静态方法，它们的返回类型为 Collector。这些方法包括 toList、toSet、toMap 以及 toCollection，相关讨论请参见其他章节。实现 Collector 的类的实例作为 Stream.collect 方法的参数。如例 4-34 所示，evenLengthStrings 方法传入字符串参数，并返回仅包含偶数长度字符串的 List。

例 4-34　利用 collect 方法返回 List

```
public List<String> evenLengthStrings(String... strings) {
    return Stream.of(strings)
        .filter(s -> s.length() % 2 == 0)
        .collect(Collectors.toList());  ❶
}
```

❶ 将偶数长度的字符串收集到 List 中

编写自定义收集器的过程则略显复杂。收集器使用 5 个函数，它们的作用是将条目累加到可变容器，并有选择性地对结果进行转换。这 5 个函数是 supplier、accumulator、combiner、finisher 以及 characteristics。

我们首先讨论 characteristics 函数，表示 Collector.Characteristics 枚举的一个不可变的元素 Set。三个枚举常量为 CONCURRENT、IDENTITY_FINISH 与 UNORDERED。CONCURRENT 表示结果容器支持多个线程在结果容器上并发地调用累加器函数，UNORDERED 表示集合操作无须保留元素的出现顺序（encounter order），IDENTITY_FINISH 表示终止器函数返回其参数而不做任何修改。

请注意，如果默认值就是实际需要的，则不必提供任何特性。

下面列出了每种参数的用途。

supplier()
: 使用 Supplier<A> 创建累加器容器（accumulator container）。

accumulator()
: 使用 BiConsumer<A,T> 为累加器容器添加一个新的数据元素。

combiner()
: 使用 BinaryOperator<A> 合并两个累加器容器。

finisher()
: 使用 Function<A,R> 将累加器容器转换为结果容器。

characteristics()
: 从枚举值中选择的 Set<Collector.Characteristics>。

如果读者熟悉 java.util.function 包定义的函数式接口，则不难理解各个参数的含义：Supplier 用于创建累加临时结果所用的容器；BiConsumer 用于将一个元素添加到累加器；BinaryOperator 表示输入类型和输出类型相同，因此可以将两个累加器合二为一；最后，Function 将累加器转换为所需的结果容器。

程序在收集过程中调用上述方法，它们由 Stream.collect 这样的方法触发。从概念上讲，集合过程相当于例 4-35 所示的（泛型）代码（取自 Javadoc）。

例 4-35　各种 Collector 方法的用法

```
R container = collector.supplier.get();            ❶
for (T t : data) {
    collector.accumulator().accept(container, t); ❷
}
return collector.finisher().apply(container);     ❸
```

❶ 创建累加器容器

❷ 将每个元素添加到累加器容器

❸ 通过 finisher 函数将累加器容器转换为结果容器

本例并未出现 combiner 函数，这或许令人感到困惑。如果处理的是顺序流，则不需要该函数，算法将既定方式执行。如果处理的是并行流，流将被分为多个子流，每个子流都会生成各自的累加器容器。接下来，在连接过程中使用 combiner 函数将多个累加器容器合并为一个，然后应用 finisher 函数。

例 4-36 显示了与例 4-34 类似的代码。

例 4-36　利用 collect 方法返回不可修改的 SortedSet

```
public SortedSet<String> oddLengthStringSet(String... strings) {
        Collector<String, ?, SortedSet<String>> intoSet =
                Collector.of(TreeSet<String>::new,          ❶
                        SortedSet::add,                     ❷
                        (left, right) -> {                  ❸
                              left.addAll(right);
                              return left;
                        },
                        Collections::unmodifiableSortedSet); ❹
        return Stream.of(strings)
                .filter(s -> s.length() % 2 != 0)
                .collect(intoSet);
    }
```

❶ Supplier：创建新的 TreeSet

❷ BiConsumer：将每个字符串添加到 TreeSet

❸ BinaryOperator：将两个 SortedSet 实例合二为一

❹ finisher：创建不可修改的 Set

程序将输出一个经过排序且不可修改的字符串集，它按字典序排序。

本例展示了如何通过 Collector.of 方法生成收集器。of 方法包括以下两种形式：

```
static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier,
    BiConsumer<A,T> accumulator,
    BinaryOperator<A> combiner,
    Function<A,R> finisher,
    Collector.Characteristics... characteristics)
```

```
static <T,R> Collector<T,R,R>    of(Supplier<R> supplier,
    BiConsumer<R,T> accumulator,
    BinaryOperator<R> combiner,
    Collector.Characteristics... characteristics)
```

`Collectors` 类提供了多种用于生成收集器的便利方法，用户几乎不需要创建自定义收集器，不过掌握相关的知识仍然很有必要。综合应用 `java.util.function` 包定义的函数式接口，可以创建各种有趣的对象。

另见

有关 `finisher` 函数（一种下游收集器）的详细讨论请参见范例 4.6，有关 `Supplier`、`Function`、`BinaryOperator` 等函数式接口的讨论请参见第 2 章，有关 `Collectors` 类定义的各种静态工具方法请参见范例 4.2。

第 5 章

流式操作、lambda表达式与方法引用的相关问题

前几章介绍了 lambda 表达式和方法引用的基础知识，并讨论了它们在流中的应用，然而在实际开发中可能会遇到一些问题。例如，既然接口提供默认方法，那么当一个类实现了具有相同默认方法签名、但具有不同实现的多个接口时会出现什么情况呢？能否在 lambda 表达式内部编写代码，但尝试访问或修改在其外部定义的变量呢？当无法为方法签名添加 `throws` 子句时，如何在 lambda 表达式中处理异常呢？

这一章将讨论上述问题。

5.1 java.util.Objects类

问题

用户希望使用静态工具方法实现非空验证、比较等操作。

方案

使用 Java 7 引入的 `java.util.Objects` 类，它在流处理中相当有用。

讨论

Java 7 引入的 `Objects` 类是一种不甚知名的类，它定义了处理各种任务所需的静态方法。

`static boolean deepEquals(Object a, Object b)`

验证两个数组是否深层相等（deep equality），该方法在数组比较中尤其有用。

`static boolean equals(Object a, Object b)`

验证两个参数是否彼此相等，它是空安全（null safe）的。

`static int hash(Object... values)`

为输入值序列生成散列码（hash code）。

`static String toString(Object o)`

如果参数不为 `null` 则返回调用 `toString` 的结果，否则返回 `null`。

`static String toString(Object o, String nullDefault)`

如果第一个参数不为 `null`，返回调用 `toString` 的结果；如果第一个参数为 `null`，返回第二个参数。

此外，可以通过 `requireNonNull` 方法的各种重载形式来验证参数。

`static <T> T requireNonNull(T obj)`

如果参数不为 `null` 则返回 `T`，否则抛出 `NullPointerException`。

`static <T> T requireNonNull(T obj, String message)`

与上一个方法相同，但由于参数为 `null` 而抛出的 `NullPointerException` 将显示指定的消息。

`static <T> T requireNonNull(T obj, Supplier<String> messageSupplier)`

与上一个方法相同，但如果第一个参数为 `null`，则调用给定的 `Supplier` 为 `NullPointerException` 生成消息。

可以看到，最后一个方法传入 `Supplier<String>` 作为参数，这也是为什么本书会讨论 `Objects` 类的原因。不过，更有说服力的原因在于 Java 8 为 `Objects` 类引入的 `isNull` 和 `nonNull` 方法，二者返回布尔值。

`static boolean isNull(Object obj)`

如果提供的引用为 `null` 则返回 `true`，否则返回 `false`。

`static boolean nonNull(Object obj)`

如果提供的引用不为 `null` 则返回 `true`，否则返回 `false`。

上述方法的优点在于，它们可以用作筛选器的 `Predicate` 实例。

我们以返回集合的类为例进行说明。例 5-1 定义了两个方法，分别返回完整的集合（无论集合的性质如何）与滤掉空元素的集合。

例 5-1　返回集合与滤掉空元素

```
List<String> strings = Arrays.asList(
    "this", null, "is", "a", null, "list", "of", "strings", null);

List<String> nonNullStrings = strings.stream()
    .filter(Objects::nonNull)       ❶
    .collect(Collectors.toList());
```

❶ 滤掉空元素

如例 5-2 所示，可以使用 `Objects.deepEquals` 方法进行测试。

例 5-2　筛选器测试

```
@Test
public void testNonNulls() throws Exception {
    List<String> strings =
        Arrays.asList("this", "is", "a", "list", "of", "strings");
    assertTrue(Objects.deepEquals(strings, nonNullStrings);
}
```

我们将上述过程一般化，使之不仅适用于字符串。例 5-3 所示的代码可以从任何列表中滤掉空元素。

例 5-3　从泛型列表（generic list）中滤掉空元素

```
public <T> List<T> getNonNullElements(List<T> list) {
    return list.stream()
            .filter(Objects::nonNull)
            .collect(Collectors.toList());
}
```

可以看到，对于一个生成 `List`（其中多个元素为空）的方法，通过上述代码滤掉其中的空元素并非难事。

5.2　lambda表达式与效果等同于`final`的变量

问题

用户希望从 lambda 表达式内部访问在其外部定义的变量。

方案

必须将在 lambda 表达式内部访问的局部变量声明为 `final`，或使其具备等同于 `final` 的效果（effectively final）。可以对特性（attribute）进行访问和修改。

讨论

20 世纪 90 年代末，在 Java 初登舞台时，开发人员偶尔会使用 GUI 库 Swing[1] 来编写客户端 Java 应用程序。与所有 GUI 库一样，Swing 组件也是事件驱动的。换言之，组件产生事件，监听器（listener）对事件做出响应。

为每个组件创建单独的监听器被视为一种良好实践，因此监听器通常作为匿名内部类实现。使用内部类不仅有助于保持程序的模块化，内部类中的代码还可以访问并修改外部类的私有特性。例如，`JButton` 实例生成 `ActionEvent`，而 `ActionListener` 接口包含一个名为

注 1：一种基于 Java 的跨平台 MVC 框架，采用单线程模型，属于 JFC 的一部分。——译者注

actionPerformed 的单一抽象方法。一旦实现被注册为监听器，就会调用该方法。相关示例如例 5-4 所示。

例 5-4　简单的 Swing GUI

```
public class MyGUI extends JFrame {
    private JTextField name = new JTextField("Please enter your name");
    private JTextField response = new JTextField("Greeting");
    private JButton button = new JButton("Say Hi");

    public MyGUI() {
        // 无关的GUI设置代码
        String greeting = "Hello, %s!";                     ❶
        button.addActionListener(new ActionListener() {
           @Override
           public void actionPerformed(ActionEvent e) {
              response.setText(
                  String.format(greeting, name.getText());  ❷
              // greeting = "Anything else";                ❸
           }
        });
    }
}
```

❶ 局部变量

❷ 访问局部变量和特性

❸ 修改局部变量（无法编译）

在本例中，greeting 字符串是在构造函数内部定义的局部变量；name 和 response 变量是类的特性；ActionListener 接口以匿名内部类的形式实现，其中一个方法为 actionPerformed。请注意，内部类中的代码：

- 可以访问特性（如 name 和 response）
- 可以修改特性（本例没有展示）
- 可以访问局部变量（greeting）
- **无法**修改局部变量

在 Java 8 之前，编译器要求 greeting 变量被声明为 final。而在 Java 8 中，变量不必采用 final 修饰，但必须具备**等同于 final 的效果**。换言之，任何试图修改局部变量值的代码都不会被编译。

当然，在 Java 8 中，应采用 lambda 表达式替换匿名内部类，如例 5-5 所示。

例 5-5　监听器的 lambda 表达式

```
String greeting = "Hello, %s!";
button.addActionListener(e ->
    response.setText(String.format(greeting,name.getText())));
```

同样地，greeting 变量不必被声明为 final，但必须具备等同于 final 的效果，否则代码无法编译。

如果读者对 Swing 示例不感兴趣，我们再来讨论另外一个示例。如例 5-6 所示，我们希望对给定 `List` 中的所有值求和。

例 5-6　对 `List` 中的所有值求和

```
List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9);

int total = 0;                                ❶
for (int n : nums) {                          ❷
    total += n;
}

total = 0;
nums.forEach(n -> total += n);                ❸

total = nums.stream()                         ❹
            .mapToInt(Integer::valueOf)
            .sum()
```

❶ 局部变量 `total`

❷ 传统的 for-each 循环

❸ 修改 lambda 表达式中的局部变量（无法编译）

❹ 将流转换为 `IntStream` 并调用 `sum` 方法

上述代码声明了一个名为 `total` 的局部变量，并采用传统的 for-each 循环对所有值求和。

`Iterable` 接口定义的 `forEach` 方法传入 `Consumer` 作为参数。如果 `Consumer` 试图修改 `total` 变量，则代码不会编译。

当然，解决这个问题的正确方式是将流转换为 `IntStream`。由于它定义了 `sum` 方法，不会涉及任何局部变量。

严格来说，函数以及在其环境中定义的可访问变量称为**闭包**（closure）。从这一定义来看，Java 对局部变量的处理并不是很明确：虽然可以访问局部变量，但无法修改。在 Java 8 中，lambda 表达式是通过值（而非变量）来关闭的，读者或许认为 lambda 表达式实际上属于闭包[2]。

另见

其他语言对闭包变量（closure variable）的处理有所不同。例如，Groovy 允许对闭包变量进行修改，但通常不认为这是一种良好实践。

注 2：那么，为什么不将 Java 8 引入的 lambda 表达式称为闭包呢？根据 Bruce Eckel 的说法，原因在于“闭包”这个术语的应用过于频繁，因而引发了争议。“当人们讨论**真正**的**闭包**时，往往意味着**他们在讨论第一种语言遇到的闭包**”。感兴趣的读者可以参考 Bruce 的博文“Are Java 8 Lambdas Closures?”。

5.3 随机数流

问题

用户希望在指定范围内生成整型、长整型或双精度随机数流。

方案

使用 `java.util.Random` 类定义的 `ints`、`longs` 与 `doubles` 方法。

讨论

如果仅需要生成一个双精度随机数，则不妨采用静态 `Math.random` 方法，它返回一个位于 0.0 和 1.0 之间的双精度值 [3]。这个过程相当于将 `java.util.Random` 类实例化并调用 `nextDouble` 方法。

`Random` 类定义了一个用于指定随机种子的构造函数。指定的种子相同，生成的随机数序列也相同，这在测试中相当有用。

如果需要生成随机数顺序流，可以使用 `Random` 类引入的 `ints`、`longs` 与 `doubles` 方法，三者的签名如下（未显示相应的重载形式）：

```
IntStream    ints()
LongStream   longs()
DoubleStream doubles()
```

借由三种方法的重载形式，我们可以指定结果流的大小以及生成数的最小值和最大值。以 `doubles` 方法为例：

```
DoubleStream doubles(long streamSize, double randomNumberOrigin,
    double randomNumberBound)
```

返回流生成给定数量（`streamSize`）的双精度伪随机数，每个数大于或等于 `randomNumberOrigin`，且严格小于 `randomNumberBound`。

如果未指定 `streamSize`，方法将返回一个所谓的“有效无限流”（effectively unlimited stream）[4]。

如果不指定最小值或最大值，对于 `doubles` 方法，最小值默认为 0，最大值默认为 1；对于 `ints` 方法，最小值和最大值默认为整型数据的完整范围；对于 `longs` 方法，最小值和最大值默认为长整型数据的（有效）完整范围。就上述三种情况而言，结果相当于重复调用 `nextDouble`、`nextInt` 与 `nextLong`。

示例代码如例 5-7 所示。

注 3：根据 Javadoc 的描述，返回值在指定范围内是“伪随机且（近似）均匀分布的”（pseudorandomly with (approximately) uniform distribution）。这表明，在讨论随机数生成器时必须做好两手准备。

注 4：可以将流视为无限，但从技术上讲它是有限的。这里的“无限”相当于 `Long.MAX_VALUE`。——译者注

例 5-7　生成随机数流

```
Random r = new Random();
r.ints(5)                          ❶
 .sorted()
 .forEach(System.out::println);

r.doubles(5, 0, 0.5)               ❷
 .sorted()
 .forEach(System.out::println);

List<Long> longs = r.longs(5)
                    .boxed()       ❸
                    .collect(Collectors.toList());
System.out.println(longs);

List<Integer> listOfInts = r.ints(5, 10, 20)
    .collect(LinkedList::new, LinkedList::add, LinkedList::addAll); ❹
System.out.println(listOfInts);
```

❶ 五个随机整数

❷ 五个位于 0（包括）和 0.5（不包括）之间的双精度随机数

❸ 将 `long` 装箱为 `Long` 以便收集

❹ 使用 `collect` 而非调用 `boxed`

如果在创建基本数据类型集合时遇到问题，后两个代码段给出了解决方案。我们无法对基本数据类型集合调用 `collect(Collectors.toList())`，相关讨论请参见范例 3.2。该范例建议，既可以通过 `boxed` 方法将 `long` 型数据转换为 `Long` 的实例，也可以使用 `collect` 方法的三参数形式并自行指定 `Supplier`、累加器与组合器。

值得注意的是，`SecureRandom` 属于 `Random` 的子类，它提供一个加密的强随机数生成器（cryptographically strong random number generator）。`Random` 类定义的 `ints`、`longs` 与 `doubles` 方法（以及它们的重载形式）同样适用于 `SecureRandom` 类，区别仅在于所用的生成器不同。

另见

有关 `boxed` 方法在 `Stream` 中的应用请参见范例 3.2。

5.4　Map接口的默认方法

问题

如果 `Map` 中包含元素，用户希望替换元素；如果 `Map` 中没有元素，用户希望添加元素；此外，用户还希望执行其他相关操作。

方案

使用 java.util.Map 接口新增的各种默认方法，如 computeIfAbsent、computeIfPresent、replace、merge 等。

讨论

从 Java 1.2 引入集合框架（collections framework）起，Map 接口就已存在。Java 8 为 Map 接口引入了一些新的默认方法，如表 5-1 所示。

表5-1：Map接口定义的默认方法

方法	描述
Compute	根据现有的键和值计算新的值
computeIfAbsent	如果键存在，返回对应的值，否则通过提供的函数计算新的值并保存
computeIfPresent	计算新的值以替换现有的值
forEach	对 Map 进行迭代，将所有键和值传递给 Consumer
getOrDefault	如果键在 Map 中存在，返回对应的值，否则返回默认值
merge	如果键在 Map 中不存在，返回提供的值，否则计算新的值
putIfAbsent	如果键在 Map 中不存在，将其关联到给定的值
remove	如果键的值与给定的值匹配，删除该键的条目
replace	将现有键的值替换为新的值
replaceAll	将 Map 中每个条目的值替换为对当前条目调用给定函数后的结果

Java 8 为已有十多年历史的 Map 接口引入了不少新方法，某些方法能为开发提供极大的便利。

1. computeIfAbsent

computeIfAbsent 方法的完整签名如下：

```
V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
```

在创建方法调用结果的缓存时，computeIfAbsent 尤其有用。我们以经典的斐波那契数递归计算为例进行讨论。如例 5-8 所示，任何大于 1 的斐波那契数等于前两个斐波那契数之和[5]。

例 5-8　斐波那契数递归计算

```
long fib(long i) {
    if (i == 0) return 0;
    if (i == 1) return 1;
    return fib(i - 1) + fib(i - 2);  ❶
}
```

❶ 效率极低

上述代码的问题在于需要进行大量重复的计算（如 fib(5) = fib(4) + fib(3) = fib(3) + fib(2) + fib(2) +fib(1) = ...），导致程序效率极低。可以利用缓存解决这个问题，函数式编程将这种技术称为**记忆化**（memoization）。如例 5-9 所示，我们将结果修改为存储 BigInteger 实例。

注 5：大部分读者想必都听过这个笑话：“据说今年的斐波那契会议将和前两年一样好。”

例 5-9　利用缓存计算斐波那契数

```
private Map<Long, BigInteger> cache = new HashMap<>();

public BigInteger fib(long i) {
    if (i == 0) return BigInteger.ZERO;
    if (i == 1) return BigInteger.ONE;

    return cache.computeIfAbsent(i, n -> fib(n - 2).add(fib(n - 1))); ❶
}
```

❶ 如果键的值在缓存中存在，返回对应的值，否则计算新的值并保存

本例采用缓存计算斐波那契数，其中键为提供的数字，值为相应的斐波那契数。`computeIfAbsent` 方法在缓存中搜索给定的数字，存在则返回对应的值，否则使用提供的 `Function` 计算新的值，将其保存在缓存中并返回。对单一方法而言，这已是很大的改进。

2. computeIfPresent

`computeIfPresent` 方法的完整签名如下：

```
V computeIfPresent(K key,
    BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

仅当与某个值关联的键在 `Map` 中存在时，`computeIfPresent` 才会更新该值。假设我们需要解析一个文本，并计算文本中每个单词的出现次数。这种一致性（concordance）计算在实际中并不鲜见。如果仅对某些特定单词感兴趣，可以使用 `computeIfPresent` 方法进行更新，如例 5-10 所示。

例 5-10　仅更新特定单词的出现次数

```
public Map<String,Integer> countWords(String passage, String... strings) {
    Map<String, Integer> wordCounts = new HashMap<>();

    Arrays.stream(strings).forEach(s -> wordCounts.put(s, 0)); ❶

    Arrays.stream(passage.split(" ")).forEach(word ->          ❷
        wordCounts.computeIfPresent(word, (key, val) -> val + 1));

    return wordCounts;
}
```

❶ 将特定单词置于映射中，并将计数器设置为 0

❷ 读取文本，仅更新特定单词的出现次数

通过将特定单词置于映射中并将初始计数器设置为 0，就能让 `computeIfPresent` 方法只更新这些值。

如例 5-11 所示，对一段文本以及一个逗号分隔的单词列表执行上述程序，可以得到所需的结果。

例 5-11　调用 countWords 方法

```
String passage = "NSA agent walks into a bar. Bartender says, " +
    "'Hey, I have a new joke for you.' NSA agent says, 'heard it'.";
```

```
Map<String, Integer> counts = demo.countWords(passage, "NSA", "agent", "joke");
counts.forEach((word, count) -> System.out.println(word + "=" + count));

// 输出为：NSA=2, agent=2, joke=1
```

可以看到，仅当所需单词是映射中的键时，程序才会更新它们的出现次数。与之前一样，采用 `Map` 接口定义的默认方法 `forEach` 打印值，该方法传入 `BiConsumer`，其参数为键和值。

3. 其他方法

`replace` 方法的用法与 `put` 方法类似，前提是键已经存在。如果键不存在，`replace` 不会执行任何操作，而 `put` 将添加一个空键（null key），不过这可能并非如我们所愿。

`replace` 方法包括两种重载形式：

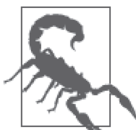
```
V replace(K key, V value)
boolean replace(K key, V oldValue, V newValue)
```

对于第一种形式，如果键在映射中存在，则将其替换为对应的值；对于第二种形式，如果键的值与指定的值相等，则将其替换为新的值。

使用不存在的键调用 `Map` 接口的 `get` 方法将返回 `null`，这个令人头疼的问题可以通过 `getOrDefault` 方法解决。该方法仅返回默认值，但不会将键添加到映射中。

`getOrDefault` 方法的签名如下：

```
V getOrDefault(Object key, V defaultValue)
```



如果键在映射中不存在，`getOrDefault` 方法将返回默认值，但不会将这个键添加到映射中。

`merge` 方法非常有用，其完整签名如下：

```
V merge(K key, V value,
        BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

对于一段给定的文本，假设我们希望统计所有单词（而不仅是特定单词）的出现次数，那么通常需要考虑两种情况：如果单词已经在映射中，则更新计数器；如果单词不在映射中，则将其置于映射中并使计数器加 1。可以通过 `merge` 方法简化这个过程，如例 5-12 所示。

例 5-12　`merge` 方法的应用

```
public Map<String, Integer> fullWordCounts(String passage) {
    Map<String, Integer> wordCounts = new HashMap<>();
    String testString = passage.toLowerCase().replaceAll("\\W"," "); ❶

    Arrays.stream(testString.split("\\s+")).forEach(word ->
        wordCounts.merge(word, 1, Integer::sum));                   ❷

    return wordCounts;
}
```

❶ 将字符串转换为小写字母并删除标点符号

❷ 更新给定单词的计数器

`merge` 方法传入键和默认值。如果键在映射中不存在，则插入默认值，否则根据原有值并使用 `BinaryOperator`（本例为 `Integer::sum`）计算出新的值。

本范例讨论了 `Map` 接口新增的默认方法，希望这些方法能为程序开发带来便利。

5.5 默认方法冲突

问题

一个类实现了两个接口，每个接口包含的默认方法相同，但实现不同。

方案

在类中实现方法。借由关键字 `super`，实现仍然可以使用接口提供的默认方法。

讨论

Java 8 支持在接口中使用静态和默认方法。默认方法提供由类继承的实现，这使得接口可以在不破坏现有类实现的情况下添加新的方法。

由于一个类可以实现多个接口，它既可能继承具有相同签名但实现不同的默认方法，也可能已包含自己的默认方法。

此时需要考虑以下三种情况。

- 如果类的方法和接口的默认方法发生冲突，则类的方法始终优先。
- 如果两个接口（其中一个接口是另一个的后代）发生冲突，则后代接口优先；如果两个类（其中一个类是另一个的后代）发生冲突，则后代类优先。
- 如果两个默认方法之间不存在继承关系，则类无法编译。

对于第三种情况，只需在类中实现方法即可，第三种情况将简化为第一种情况。

观察例 5-13 所示的 `Company` 接口和例 5-14 所示的 `Employee` 接口。

例 5-13　包含默认方法的 `Company` 接口

```
public interface Company {
    default String getName() {
        return "Initech";
    }

    // 其他方法
}
```

关键字 `default` 将 `getName` 方法指定为默认方法，它提供一个返回企业名称的实现。

例 5-14　包含默认方法的 Employee 接口

```
public interface Employee {
    String getFirst();

    String getLast();

    void convertCaffeineToCodeForMoney();

    default String getName() {
        return String.format("%s %s", getFirst(), getLast());
    }
}
```

Employee 接口同样定义了一个名为 getName 的默认方法，其签名与 Company 接口中的 getName 方法相同，但二者的实现不同。如例 5-15 所示，CompanyEmployee 类实现了 Company 和 Employee 两个接口，从而导致冲突。

例 5-15　最初的 CompanyEmployee 类（无法编译）

```
public class CompanyEmployee implements Company, Employee {
    private String first;
    private String last;

    @Override
    public void convertCaffeineToCodeForMoney() {
        System.out.println("Coding...");
    }

    @Override
    public String getFirst() {
        return first;
    }

    @Override
    Public String getLast() {
        return last;
    }
}
```

由于 CompanyEmployee 类继承了与 getName 无关的默认方法，它无法编译。为解决这个问题，需要在 CompanyEmployee 类中添加用户自定义的 getName 方法，它将重写两个默认方法。

不过，借由关键字 super，仍然可以使用所提供的默认方法，如例 5-16 所示。

例 5-16　调整后的 CompanyEmployee 类

```
public class CompanyEmployee implements Company, Employee {

    @Override
    public String getName() {                                    ❶
        return String.format("%s working for %s",
            Employee.super.getName(), Company.super.getName()); ❷
    }

    // 其余代码和之前一样
}
```

❶ 实现 getName 方法

❷ 通过 super 访问默认实现

可以看到，CompanyEmployee 类中的 getName 方法根据 Company 和 Employee 接口定义的两个默认方法 getName 构建了一个 String。

最好的消息是，这与默认方法一样复杂。读者现在已经了解了如何处理默认方法冲突。

实际上，我们还要考虑一种极端情况。如果 Company 接口定义了 getName 方法但没有将其指定为 default（也不存在相应的实现，即 getName 成为抽象方法），那么当 Employee 接口也定义了 getName 方法时，是否还会导致冲突呢？答案是肯定的。有趣的是，仍然需要在 CompanyEmployee 类中提供一个实现。

在 Java 8 之前，如果两个接口包含相同的方法且没有指定为默认方法，这并不会导致冲突，但类必须提供一个实现。

另见

有关接口中默认方法的讨论请参见范例 1.5。

5.6　集合与映射的迭代

问题

用户希望对集合或映射进行迭代。

方案

使用 java.lang.Iterable 或 java.util.Map 接口新增的默认方法 forEach。

讨论

除了使用循环对线性集合（即实现 Collection 或其后代的类）进行迭代外，可以通过 Iterable 接口引入的默认方法 forEach 实现同样的目的。

根据 Javadoc 的描述，forEach 方法的签名为：

```
default void forEach(Consumer<? super T> action)
```

forEach 方法的参数为 Consumer，它是 java.util.function 包引入的一种函数式接口，表示传入一个泛型参数（generic argument）且不返回任何结果的操作。根据 Javadoc 的描述，“Consumer 与大部分函数式接口不同，它在执行时会产生副作用”。



纯函数（pure function）在执行时不会产生副作用。换言之，如果参数相同，则每次返回的结果也相同。函数式编程将其称为**引用透明性**（referential transparency），即函数可以被它的值所替换。

java.util.Collection 是 Iterable 的子接口，因此 forEach 方法适用于从 ArrayList 到 LinkedHashSet 的所有线性集合，这使得线性集合的迭代易如反掌（如例 5-17 所示）。

例 5-17　对线性集合进行迭代

```
List<Integer> integers = Arrays.asList(3, 1, 4, 1, 5, 9);

integers.forEach(new Consumer<Integer>() {     ❶
    @Override
    public void accept(Integer integer) {
        System.out.println(integer);
    }
});

integers.forEach((Integer n) -> {              ❷
    System.out.println(n);
});

integers.forEach(n -> System.out.println(n)); ❸

integers.forEach(System.out::println);         ❹
}
```

❶ 匿名内部类实现

❷ 完整形式的 lambda 代码块

❸ lambda 表达式

❹ 方法引用

在本例中，匿名内部类只是显示为 Consumer 接口的 accept 方法的签名。观察内部类可知，accept 方法传入一个参数并返回 void，本例所用的 lambda 表达式与之兼容。由于两个 lambda 表达式都包含对 System.out.println 方法的单次调用，forEach 方法可以用作方法引用。

Map 接口同样引入 forEach 方法作为默认方法，它传入 BiConsumer：

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

BiConsumer 也是 java.util.function 包新增的一种接口，表示一个传入两个泛型参数并返回 void 的函数。在实现 Map 接口的 forEach 方法时，参数是 entrySet 方法中 Map.Entry 实例的键和值。

换言之，对 Map 进行迭代现在就像对 List、Set 或其他任何线性集合进行迭代一样简单。示例代码如例 5-18 所示。

例 5-18　对 Map 进行迭代

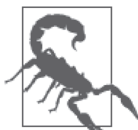
```
Map<Long, String> map = new HashMap<>();
map.put(86L, "Don Adams (Maxwell Smart)");
map.put(99L, "Barbara Feldon");
map.put(13L, "David Ketchum");
map.forEach((num, agent) ->
    System.out.printf("Agent %d, played by %s%n", num, agent));
```

输出如例 5-19[6] 所示。

例 5-19　对 Map 迭代后的输出

```
Agent 99, played by Barbara Feldon
Agent 86, played by Don Adams (Maxwell Smart)
Agent 13, played by David Ketchum
```

在 Java 8 之前，为了实现对 `Map` 的迭代，需要首先通过 `keySet` 方法获取键的 `Set`，或通过 `entrySet` 方法获取 `Map.Entry` 实例。而 Java 8 引入的默认方法 `forEach` 使迭代操作得以简化。



跳出 for-each 循环并非易事，这一点请谨记在心。为解决这个问题，考虑将流处理代码重写为 `filter` 或 `sorted`，并后跟 `findFirst`。

另见

有关函数式接口 `Consumer` 和 `BiConsumer` 的讨论请参见范例 2.1。

5.7　利用Supplier创建日志消息

问题

用户希望创建由日志级别（log level）控制是否可见的日志消息。

方案

使用 `java.util.logging.Logger` 类新增的各种日志方法，它们传入 `Supplier` 作为参数。

讨论

目前，`Logger` 类中的日志方法（如 `info`、`warning`、`severe` 等）包括两种重载形式，一种传入单个 `String` 作为参数，另一种传入 `Supplier<String>` 作为参数。

例 5-20 显示了各种日志方法的签名。[7]

例 5-20　Logger 类中各种日志方法的重载形式

```
void config(String msg)
void config(Supplier<String> msgSupplier)
```

注 6：例 5-18 和例 5-19 取自美国经典电视剧《糊涂侦探》，这部由梅尔 • 布鲁克斯（Mel Brooks）和巴克 • 亨利（Buck Henry）编导的间谍喜剧在 1965 年到 1970 年间播出。主角麦克斯韦 • 史马特（Maxwell Smart）糅合了詹姆斯 • 邦德与雅克 • 克鲁索（英国电影《糊涂大侦探》主角）的特点，又称特工 86 号（Agent 86）。他的女搭档是特工 99 号（Agent 99），同事是特工 13 号（Agent 13）。

注 7：读者或许奇怪，Java 日志框架的设计者为什么不采用与其他日志 API 相同的日志级别（Trace、Debug、Info、Warn、Error、Fatal）。这是一个非常好的问题，如果读者找到答案，也请告诉作者。

```
void fine(String msg)
void fine(Supplier<String> msgSupplier)

void finer(String msg)
void finer(Supplier<String> msgSupplier)

void finest(String msg)
void finest(Supplier<String> msgSupplier)

void info(String msg)
void info(Supplier<String> msgSupplier)

void warning(String msg)
void warning(Supplier<String> msgSupplier)

void severe(String msg)
void severe(Supplier<String> msgSupplier)
```

每种方法的第一种重载形式（传入 `String`）是 Java 1.4 引入的，而第二种重载形式（传入 `Supplier`）是 Java 8 引入的，它在标准库中的实现如例 5-21 所示。

例 5-21　`Logger` 类的实现细节

```
public void info(Supplier<String> msgSupplier) {
    log(Level.INFO, msgSupplier);
}

public void log(Level level, Supplier<String> msgSupplier) {
    if (!isLoggable(level)) {                                ❶
        return;
    }
    LogRecord lr = new LogRecord(level, msgSupplier.get()); ❷
    doLog(lr);
}
```

❶ 如果不显示消息则返回

❷ 调用 `get` 方法以便在 `Supplier` 中检索消息

上述实现并非构建一个永远不会显示的消息，而是检查消息是否是“可记录的”（loggable）。如果所提供的消息是一个简单的字符串，程序将评估它是否已被记录。在上述日志方法中，第二种重载形式（传入 `Supplier`）支持在消息前添加空括号和箭头（`() ->`）以将其转换为 `Supplier`，且仅当日志级别合适时才会调用它。例 5-22 显示了 `info` 方法的两种重载形式。

例 5-22　在 `info` 方法中使用 `Supplier`

```
private Logger logger = Logger.getLogger(this.getClass().getName());
private List<String> data = new ArrayList<>();

// 用数据填充列表

logger.info("The data is " + data.toString());       ❶
logger.info(() -> "The data is " + data.toString()); ❷
```

❶ 无论是否显示 Info 消息，都会构建参数

❷ 仅当日志级别显示 Info 消息时，才会构建参数

可以看到，消息在列表的每个对象上调用 `toString` 方法。在第一条 `logger.info` 语句中，无论程序是否显示 Info 消息，都会创建结果字符串；在第二条 `logger.info` 语句中，在消息前添加 `() ->` 就能将日志参数转换为 `Supplier`，这意味着只有使用消息时才会调用 `Supplier` 的 `get` 方法。

采用相同类型的 `Supplier` 替换参数的技术称为**延迟执行**（deferred execution），可以在任何对象创建成本较高的上下文中使用。

另见

延迟执行是 `Supplier` 的主要用例之一，有关 `Supplier` 接口的讨论请参见范例 2.2。

5.8 闭包复合

问题

用户希望连续应用一系列简单且独立的函数。

方案

使用 `Function`、`Consumer` 与 `Predicate` 接口中定义为默认方法的复合方法（composition method）。

讨论

函数式编程的优点之一在于支持创建若干简单、可重复使用的函数，将这些函数组合在一起就能解决复杂的问题。为此，`java.util.function` 包引入的函数式接口定义了各种有助于简化复合操作的方法。

例如，`Function` 接口包括 `compose` 和 `andThen` 两种默认方法，二者的签名如例 5-23 所示。

例 5-23　`Function` 接口定义的复合方法

```
default <V> Function<V,R>       compose(Function<? super V,? extends T> before)
default <V> Function<T,V>       andThen(Function<? super R,? extends V> after)
```

从方法签名中的哑元（dummy argument）不难看出两种方法的作用：`compose` 方法在原始函数**之前**应用其参数，而 `andThen` 方法在原始函数**之后**应用其参数。

为说明两种方法的应用，考虑例 5-24 所示的简单示例。

例 5-24　`compose` 和 `andThen` 方法的应用

```
Function<Integer, Integer> add2  = x -> x + 2;
Function<Integer, Integer> mult3 = x -> x * 3;
```

```
Function<Integer, Integer> mult3add2  = add2.compose(mult3); ❶
Function<Integer, Integer> add2mmult3 = add2.andThen(mult3); ❷

System.out.println("mult3add2(1): " + mult3add2.apply(1));
System.out.println("add2mult3(1): " + add2mult3.apply(1));
```

❶ 先执行 mult3 函数，再执行 add2 函数

❷ 先执行 add2 函数，再执行 mult3 函数

可以看到，add2 函数的作用是将参数加 2，而 mult3 函数的作用是将参数乘 3。通过 compose 方法创建的复合函数 mult3add2 先执行 mult3，再执行 add2；通过 andThen 方法创建的复合函数 add2mult3 则相反，先执行 add2，再执行 mult3。

两个复合函数的执行结果如下：

```
mult3add2(1): 5 // 因为(1 * 3) + 2 == 5
add2mult3(1): 9 // 因为(1 + 2) * 3 == 9
```

复合函数的结果仍然是函数，因此这个过程创建的操作可供今后使用。例如，如果收到的数据属于 HTTP 请求的一部分，说明数据是以字符串形式传输的。虽然已有可用于操作数据的方法，但前提为数据是数字。如果这种情况频繁发生，可以考虑在应用数值操作之前编写一个解析字符串数据的函数。详见例 5-25。

例 5-25　首先将字符串解析为整数，然后加 2

```
Function<Integer, Integer> add2 = x -> x + 2;
Function<String, Integer> parseThenAdd2 = add2.compose(Integer::parseInt);
System.out.println(parseThenAdd2.apply("1"));
// 打印3
```

本例创建了一个名为 parseThenAdd2 的函数，它先调用静态方法 Integer.parseInt，再将所得结果加 2。另一方面，也可以定义一个先执行数值操作，再调用 toString 方法的函数，如例 5-26 所示。

例 5-26　首先加 2，然后将数字转换为字符串

```
Function<Integer, Integer> add2 = x -> x + 2;
Function<Integer, String> plus2toString = add2.andThen(Object::toString);
System.out.println(plus2toString.apply(1));
// 打印"3"
```

上述操作返回一个函数，它传入 Integer 参数并返回 String。

如例 5-27 所示，Consumer 接口也定义了一个用于闭包复合的方法。

例 5-27　Consumer 接口定义的复合方法

```
default Consumer<T> andThen(Consumer<? super T> after)
```

根据 Javadoc 的描述，andThen 方法返回一个复合 Consumer，它依次执行原始操作和 after 指定的操作。如果执行任一操作时抛出异常，异常将被转发给组合操作的调用者。

示例代码如例 5-28 所示。

例 5-28　用于打印和记录的复合 Consumer

```
Logger log = Logger.getLogger(...);
Consumer<String> printer = System.out::println;
Consumer<String> logger = log::info;

Consumer<String> printThenLog = printer.andThen(logger);
Stream.of("this", "is", "a", "stream", "of", "strings").forEach(printThenLog);
```

程序首先创建了两个 `Consumer`，一个用于打印结果到控制台，另一个用于日志记录。接下来，程序创建了一个复合 `Consumer`，可以一次性打印并记录流的所有元素。

`Predicate` 接口定义了三种用于谓词复合的方法，如例 5-29 所示。

例 5-29　Predicate 接口定义的复合方法

```
default Predicate<T>     and(Predicate<? super T> other)
default Predicate<T>     negate()
default Predicate<T>     or(Predicate<? super T> other)
```

`and`、`or`、`negate` 方法分别使用逻辑与、逻辑或、逻辑非操作实现谓词的复合，每种方法返回一个复合 `Predicate`。

接下来，我们讨论一个关于整数的有趣问题。完全平方数（perfect square）是其平方根同样为整数的数，而三角形数（triangle number）是一定数目的点或圆在等距离排列下可以形成等边三角形的数[8]。

例 5-30 创建了两个用于计算完全平方数和三角形数的方法，并通过 `and` 方法查找既是完全平方数，又是三角形数的数。

例 5-30　既是完全平方数，又是三角形数的数

```
public static boolean isPerfect(int x) {     ❶
    return Math.sqrt(x) % 1 == 0;
}

public static boolean isTriangular(int x) {     ❷
    double val = (Math.sqrt(8 * x + 1) - 1) / 2;
    return val % 1 == 0;
}

// 其他代码

IntPredicate triangular = CompositionDemo::isTriangular;
IntPredicate perfect = CompositionDemo::isPerfect;
IntPredicate both = triangular.and(perfect);

IntStream.rangeClosed(1, 10_000)
        .filter(both)
        .forEach(System.out::println);       ❸
```

注 8：参见维基百科有关“三角形数”的介绍。在一个房间中，如果每个人只与其他人握手一次，则三角形数是握手次数的总和。（例如，房间中有 2 个人时握手次数为 1，有 3 个人时握手次数为 3，有 4 个人时握手次数为 6，有 5 个人时握手次数为 10，有 6 个人时握手次数为 15，以此类推。那么 1、3、6、10、15 就是前 5 个三角形数，第 n 个三角形数的计算公式为 $\frac{n(n+1)}{2}$。——译者注）

❶ 部分完全平方数：1、4、9、16、25、36、49、64、81……

❷ 部分三角形数：1、3、6、10、15、21、28、36、45……

❸ 既是完全平方数，又是三角形数（1 到 10 000 之间）：1、36、1225

借由复合函数，可以将若干简单的函数组合在一起以构建复杂的操作[9]。

另见

有关 `Function` 接口的讨论请参见范例 2.4，有关 `Consumer` 接口的讨论请参见范例 2.1，有关 `Predicate` 接口的讨论请参见范例 2.3。

5.9 利用提取的方法实现异常处理

问题

lambda 表达式中的代码需要抛出异常，但用户不希望将 lambda 代码块与异常处理的代码混在一起。

方案

创建一个单独的方法来执行操作、处理异常，在 lambda 表达式中调用提取的方法。

讨论

lambda 表达式实际上属于函数式接口中单一抽象方法的实现。与匿名内部类一样，lambda 表达式只能抛出在抽象方法签名中声明的异常。

如果所需的异常为非受检异常（unchecked exception），则情况相对简单。所有非受检异常均源自 `java.lang.RuntimeException` 类[10]。与其他 Java 代码类似，lambda 表达式可以抛出运行时异常（runtime exception）而不必进行声明或将代码包装在 try/catch 块中，异常随后被传送给调用者。

例 5-31 创建了一个名为 `div` 的方法，它采用常数值对集合中所有元素进行除法操作。

例 5-31　可能抛出非受检异常的 lambda 表达式

```
public List<Integer> div(List<Integer> values, Integer factor) {
    return values.stream()
        .map(n -> n / factor)  ❶
        .collect(Collectors.toList());
}
```

注 9：Unix 操作系统就是基于这种理念构建的，具有类似的优点。

注 10：这可能是整个 Java API 中最糟糕的命名——既然所有异常均在运行时抛出（否则它们属于编译器错误），那么将这个类命名为 `UncheckedException` 不是更直接吗？或许是为了凸显这个命名的糟糕，Java 8 引入了一个称为 `java.io.UncheckedIOException` 的类，以规避本范例讨论的某些问题。

❶ 可能抛出 `ArithmeticException`

对整数除法而言，除数为 0 将抛出 `ArithmeticException`（非受检异常）[11]，它将传送给调用者，如例 5-32 所示。

例 5-32　客户端代码

```
List<Integer> values = Arrays.asList(30, 10, 40, 10, 50, 90);
List<Integer> scaled = demo.div(values, 10);
System.out.println(scaled);
// 打印[3, 1, 4, 1, 5, 9]

scaled = demo.div(values, 0);
System.out.println(scaled);
// 抛出ArithmeticException（因为除数为0）
```

客户端代码调用 `div` 方法，如果除数为 0，lambda 表达式将抛出 `ArithmeticException`。客户端可以在 `map` 方法中添加一个 try/catch 块以处理异常，但代码的可读性会因此而变得很差（如例 5-33 所示）。

例 5-33　包含 try/catch 代码块的 lambda 表达式

```
public List<Integer> div(List<Integer> values, Integer factor) {
    return values.stream()
        .map( n -> {
            try {
                return n / factor;
            } catch (ArithmeticException e) {
                e.printStackTrace();
            }
        })
        .collect(Collectors.toList());
}
```

只要在函数式接口中声明受检异常（checked exception），就能应用上述过程。

一般来说，应尽量保持流处理代码的简洁，让每个中间操作占据一行。我们可以将 `map` 方法内部的函数提取到另一个方法以简化代码，然后调用这个方法来完成流处理，如例 5-34 所示。

例 5-34　将 lambda 表达式提取到另一个方法

```
private Integer divide(Integer value, Integer factor) {
    try {
        return value / factor;
    } catch (ArithmeticException e) { ❶
        e.printStackTrace();
    }
}

public List<Integer> divUsingMethod(List<Integer> values, Integer factor) {
    return values.stream()
```

注 11：有趣的是，如果将被除数和除数的类型从 `Integer` 改为 `Double`，那么即便除数为 0.0，程序也不会抛出任何异常，而是输出一个所有元素为“无穷大”（Infinity）的结果。无论读者是否相信，根据二进制计算机处理浮点数需要遵循的 IEEE 754 规范，这种操作是合法的（作者在使用 Fortran 语言编程时曾被这种情况搞得十分头疼，花了不少时间才摆脱这个噩梦）。

```
            .map(n -> divide(n, factor))  ❷
            .collect(Collectors.toList());
    }
```

❶ 这段代码负责处理异常

❷ 流处理代码得以简化

此外，如果提取的方法不需要 `factor` 值，则 `map` 方法的参数可以简化为一个方法引用。

将 lambda 表达式提取到单独的方法同样具有不少优点。我们既可以为提取的方法编写测试（对于私有方法则使用反射），也可以在方法内部设置断点，还可以使用与方法相关的其他机制。

另见

有关受检异常与 lambda 表达式的讨论请参见范例 5.10，有关利用泛型包装器处理异常的讨论请参见范例 5.11。

5.10　受检异常与lambda表达式

问题

存在一个抛出受检异常的 lambda 表达式，且所实现的函数式接口中的抽象方法并未声明该异常。

方案

为 lambda 表达式添加一个 try/catch 代码块，或委托给某个提取的方法进行处理。

讨论

lambda 表达式实际上属于函数式接口中单一抽象方法的实现，因此只能抛出在抽象方法签名中声明的受检异常。

假设我们希望通过 URL 调用服务，且需要根据字符串参数的集合构建一个查询字符串，那么参数应该采用可以在 URL 中使用的方式进行编码。为此，Java 提供了一个称为 `java.net.URLEncoder` 的类（其用途从类名中一目了然），包括一个静态方法 `encode`，它传入 `String` 作为参数，并根据指定的编码方式对其进行编码。

我们很容易写出类似例 5-35 所示的代码。

例 5-35　对字符串集合进行 URL 编码（无法编译）

```
    public List<String> encodeValues(String... values) {
        return Arrays.stream(values)
            .map(s -> URLEncoder.encode(s, "UTF-8")))  ❶
            .collect(Collectors.toList());
    }
```

❶ 抛出一个必须处理的异常 UnsupportedEncodingException

encodeValues 方法传入一个字符串的可变参数列表，并根据推荐的 UTF-8 编码方式，尝试采用 UREncoder.encode 方法对每个字符串进行编码。然而，由于 encodeValues 方法抛出受检异常 UnsupportedEncodingException，导致上述代码无法编译。

即便声明 encodeValues 方法抛出该异常，代码仍然无法编译。

例 5-36　声明异常（仍然无法编译）

```
public List<String> encodeValues(String... values)
    throws UnsupportedEncodingException {  ❶
        return Arrays.stream(values)
            .map(s -> URLEncoder.encode(s, "UTF-8")))
            .collect(Collectors.toList());
}
```

❶ 声明 encodeValues 方法抛出 UnsupportedEncodingException，但代码仍然无法编译

问题在于，从 lambda 表达式抛出异常就像采用某种方法构建一个完全独立的类，并从中抛出异常。我们可以将 lambda 表达式视为匿名内部类的实现，由此可见，内部对象抛出的异常仍然需要在内部对象而非周围对象（surrounding object）中进行处理或声明。正确的代码如例 5-37 所示，包括匿名内部类和 lambda 表达式两种实现。

例 5-37　利用 try/catch 代码块进行 URL 编码（可以编译）

```
public List<String> encodeValuesAnonInnerClass(String... values) {
    return Arrays.stream(values)
        .map(new Function<String, String>() {           ❶
            @Override
            public String apply(String s) {              ❷
                try {
                    return URLEncoder.encode(s, "UTF-8");
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                    return "";
                }
            }
        })
        .collect(Collectors.toList());
}

public List<String> encodeValues(String... values) {    ❸
    return Arrays.stream(values)
        .map(s -> {
            try {
                return URLEncoder.encode(s, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
                return "";
            }
        })
        .collect(Collectors.toList());
}
```

❶ 匿名内部类实现

❷ 包含将抛出受检异常的代码

❸ lambda 表达式实现

由于 apply 方法（Function 接口包含的单一抽象方法）并未声明任何受检异常，必须在任何实现该方法的 lambda 表达式中添加一个 try/catch 代码块。如果使用 lambda 表达式（如本例所示），我们甚至无法看到 apply 方法签名，遑论对其进行修改（程序也不允许）。

例 5-38 显示了如何利用提取的方法进行编码。

例 5-38　委托给 encodeString 方法进行 URL 编码

```
private String encodeString(String s) { ❶
    try {
        return URLEncoder.encode(s, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public List<String> encodeValuesUsingMethod(String... values) {
    return Arrays.stream(values)
        .map(this::encodeString)        ❷
        .collect(Collectors.toList());
}
```

❶ 用于异常处理的提取方法

❷ 提取方法的方法引用

上述代码有效且易于实现，并提供了一种可以分别进行测试和调试的方法。唯一的不足之处在于，我们需要为每个可能抛出异常的操作提取一个方法。但前面的范例曾经提到过，这使得对流处理的各个组件进行测试更加容易。

另见

有关利用提取的方法处理 lambda 表达式中的异常请参见范例 5.9，有关利用泛型包装器处理异常的讨论请参见范例 5.11。

5.11　泛型异常包装器的应用

问题

存在一个抛出异常的 lambda 表达式，但用户希望使用泛型包装器（generic wrapper）来捕获所有受检异常，并将它们作为非受检异常重新抛出。

方案

创建特殊的异常类（exception class），添加一个接受这些类的泛型方法，并返回不抛出异常的 lambda 表达式。

讨论

范例 5.9 和范例 5.10 讨论了如何委托一个单独的方法来处理 lambda 表达式抛出的异常，不过我们需要为每个可能抛出异常的操作定义一个私有方法。可以采用**泛型包装器**使其更为通用。

为此，我们定义一个单独的函数式接口，它包含一个声明抛出 `Exception` 的方法，然后通过某种包装器方法（wrapper method）将其连接到用户代码。

例如，`Stream` 接口的 `map` 方法需要一个 `Function`，但 `Function` 接口的 `apply` 方法不会声明任何受检异常。为了在 `map` 方法中使用可能抛出受检异常的 lambda 表达式，我们创建一个单独的函数式接口，声明它将抛出 `Exception`，如例 5-39 所示。

例 5-39　抛出 `Exception` 的函数式接口

```
@FunctionalInterface
public interface FunctionWithException<T, R, E extends Exception> {
    R apply(T t) throws E;
}
```

接下来，在 try/catch 代码块中包装 `apply` 方法，以添加一个传入 `FunctionWithException` 并返回 `Function` 的包装器方法，如例 5-40 所示。

例 5-40　用于处理异常的包装器方法

```
private static <T, R, E extends Exception>
    Function<T, R> wrapper(FunctionWithException<T, R, E> fe) {
        return arg -> {
            try {
                return fe.apply(arg);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };
}
```

如上所示，`wrapper` 方法接受抛出 `Exception` 的代码，并构建必要的 try/catch 块，同时委托给 `apply` 方法。在本例中，`wrapper` 方法被声明为 `static`，但这并非强制要求。如例 5-41 所示，我们可以使用任何抛出异常的 `Function` 来调用包装器。

例 5-41　泛型静态包装器方法的应用

```
public List<String> encodeValuesWithWrapper(String... values) {
    return Arrays.stream(values)
        .map(wrapper(s -> URLEncoder.encode(s, "UTF-8"))) ❶
        .collect(Collectors.toList());
}
```

❶ 使用 `wrapper` 方法

接下来，可以在抛出异常的 `map` 操作中编写代码，`wrapper` 方法会将其作为未受检异常重新抛出。这种方案的不足之处在于，需要为所有计划使用的函数式接口创建单独的泛型包装器（如 `ConsumerWithException` 和 `SupplierWithException`）。

这未免有些复杂。有鉴于此，不难理解为何某些 Java 框架（如 Spring 和 Hibernate）甚至整个语言（如 Groovy 和 Kotlin）在捕获所有受检异常后，再将它们作为非受检异常重新抛出。

另见

有关受检异常与 lambda 表达式的讨论请参见范例 5.10，有关利用提取的方法处理 lambda 表达式中的异常请参见范例 5.9。

第 6 章

Optional类

> 唉，为什么只要与 Optional 类有关的话题，就有那么多消息呢？
>
> ——Brian Goetz
>
> lambda-libs-spec-experts 邮件列表管理员
>
> （2013 年 10 月 23 日）

Java 8 API 引入了一种称为 `java.util.Optional<T>` 的类。虽然不少开发人员认为这个类的作用是从代码中删除 `NullPointerException`，不过其实际用途并非如此。相反，`Optional` 类设计用来在返回值可能合法为 `null` 时与用户通信。如果根据某些条件过滤值流后恰好没有元素存留，就会出现返回值合法为 `null` 的情况。

在 Stream API 中，如果流中没有元素，`reduce`、`min`、`max`、`findFirst` 以及 `findAny` 方法将返回 `Optional`。

`Optional` 实例具有两种状态，要么是对 `T` 类型实例的引用，要么为空。前者称为**存在**（present），后者称为**空**（empty，与 `null` 相对）。



虽然 `Optional` 是一种引用类型，但不应被赋值为 `null`，否则将导致严重的错误。

这一章着眼于 `Optional` 的各种习惯用法。如何正确使用 `Optional` 可能引发公司内部的激烈争论[1]，好在有一些标准建议可供参考，遵循这些原则有助于在讨论时清晰传递我们的意图。

注 1：作者对于这种讨论相当老练。

6.1 Optional的创建

问题

用户希望从现有值返回 `Optional`。

方案

使用 `Optional.of`、`Optional.ofNullable` 或 `Optional.empty` 方法。

讨论

与 Java 8 新增的众多类一样，`Optional` 实例也是不可变的（immutable）。API 将 `Optional` 称为**基于值的类**（value-based class），即 `Optional` 实例：

- 被声明为 `final` 且是不可变[2]的（但可能包含对可变对象的引用）；
- 不提供公共构造函数（public constructor），因此必须采用工厂方法加以实例化；
- 具有 `equals`、`hashCode` 与 `toString` 的实现，这些实现是通过实例的状态计算出来的。

`Optional` 与不可变性

`Optional` 实例是不可变的，但它包装的对象却不一定是不可变的。如果创建一个包含可变对象实例的 `Optional`，则仍然可以对实例进行修改，如例 6-1 所示。

例 6-1　`Optional` 是不可变的吗？

```
AtomicInteger counter = new AtomicInteger();
Optional<AtomicInteger> opt = Optional.ofNullable(counter);

System.out.println(optional);                          // Optional[0]

counter.incrementAndGet();                             ❶
System.out.println(optional);                          // Optional[1]

optional.get().incrementAndGet();                      ❷
System.out.println(optional);                          // Optional[2]

optional = Optional.ofNullable(new AtomicInteger()); ❸
```

❶ 直接使用计数器自增

❷ 检索包含的值并自增

❸ 可以对 `Optional` 引用重新赋值

注 2：有关不可变性的讨论请参见“`Optional` 与不可变性”注释。

> 我们既可以通过原始引用修改包含的值，也可以在 Optional 上调用 get 以检索某个值进行修改。我们甚至可以对引用本身重新赋值，换言之，不能将不可变性与 final 混为一谈。但是，无法对 Optional 实例本身进行修改，因为不存在能执行这种操作的方法。
>
> 对“不可变”的定义略显模糊，这在 Java 中并不鲜见。Java 并未提供一种良好的内置方式来创建只能产生无法改变的对象的类。

我们可以通过静态工厂方法 empty、of 与 ofNullable 来创建 Optional，三者的签名如下：

```
static <T> Optional<T>  empty()
static <T> Optional<T>  of(T value)
static <T> Optional<T>  ofNullable(T value)
```

显而易见，empty 方法将返回一个空 Optional。如果参数为 null，of 方法将返回一个包装指定值或抛出异常的 Optional，其应用如例 6-2 所示。

例 6-2　通过 of 方法创建 Optional

```
public static <T> Optional<T> createOptionalTheHardWay(T value) {
    return value == null ? Optional.empty() : Optional.of(value);
}
```

在本例中，createOptionalTheHardWay 之所以被命名为“The Hard Way”，并非因为它难以实现，而是存在另一种更简单的方法，这就是例 6-3 所示的 ofNullable 方法。

例 6-3　通过 ofNullable 方法创建 Optional

```
public static <T> Optional<T> createOptionalTheEasyWay(T value) {
    return Optional.ofNullable(value);
}
```

在 Java 8 的引用实现中，ofNullable 方法的实现其实就是 createOptionalTheHardWay 中的语句：检查包含的值是否为 null，若是，则返回一个空 Optional，否则使用 of 方法进行包装。

此外，包装基本数据类型的 OptionalInt、OptionalLong 与 OptionalDouble 类不能为 null，因此三者只有 of 方法，没有 ofNullable 方法。

```
static OptionalInt    of(int value)
static OptionalLong   of(long value)
static OptionalDouble of(double value)
```

请注意，三个类的 getter 方法分别为 getAsInt、getAsLong 与 getAsDouble，而不是 getInt、getLong 与 getDouble。

另见

本章其他范例（如范例 6.4 和范例 6.5）根据提供的集合（而非现有值）来创建 Optional 值，范例 6.3 使用本范例讨论的方法来包装提供的值。

6.2　从Optional中检索值

问题

用户希望从 Optional 中提取包含的值。

方案

如果确定 Optional 中存在值，则使用 get 方法，否则使用 orElse 方法的某种形式。如果只希望当值存在时才执行 Consumer，也可以使用 ifPresent 方法。

讨论

当调用一个返回 Optional 的方法时，可以通过调用 get 方法来检索其中包含的值。如果 Optional 为空，get 方法将抛出 NoSuchElementException。

接下来，我们讨论如何从一个字符串流中返回第一个偶数长度的字符串，如例 6-4 所示。

例 6-4　检索第一个偶数长度的字符串

```
Optional<String> firstEven =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();
```

由于流中的所有字符串都无法通过筛选器，findFirst 方法将返回 Optional<String>。可以通过在 Optional 上调用 get 方法来打印返回值：

```
System.out.println(firstEven.get()) // 即便这条语句可以执行，也应避免使用
```

请注意，虽然上述语句可以执行，但除非确定 Optional 中包含值，否则不要调用 get 方法，以免程序抛出异常，如例 6-5 所示。

例 6-5　检索第一个奇数长度的字符串

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();

System.out.println(firstOdd.get()); // 抛出NoSuchElementException
```

我们可以采用多种方案解决这个问题。例如，首先确定 Optional 中是否包含值，然后再检索，如例 6-6 所示。

例 6-6　检索第一个偶数长度的字符串（先确定是否包含值，再使用 get 方法）

```
Optional<String> firstEven =                                   ❶
    Stream.of("five", "even", "length", "string", "values")
          .filter(s -> s.length() % 2 == 0)
          .findFirst();

System.out.println(
    first.isPresent() ? first.get() : "No even length strings"); ❷
```

❶ 与例 6-4 所示的代码相同

❷ 仅当 `isPresent` 方法返回 `true` 时才调用 `get` 方法

不过，虽然上述代码可以执行，但只是添加了一个 `isPresent` 方法进行非空验证，程序并未有太大改进。

所幸我们还有一个更好的选择，这就是非常方便的 `orElse` 方法，如例 6-7 所示。

例 6-7　使用 `orElse` 方法

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();

System.out.println(firstOdd.orElse("No odd length strings"));
```

如果包含的值存在，则 `orElse` 方法返回该值，否则返回提供的默认值。因此，如果已有回退值（fallback value），那么采用 `orElse` 方法是相当方便的。

`orElse` 方法包括三种形式。

- 当包含的值存在时，`orElse(T other)` 将返回该值，否则返回指定的默认值 `other`。
- 当包含的值存在时，`orElseGet(Supplier<? extends T> other)` 将返回该值，否则调用 `Supplier` 并返回相应的结果。
- 当包含的值存在时，`orElseThrow(Supplier<? extends X> exceptionSupplier)` 将返回该值，否则抛出由 `Supplier` 产生的异常。

`orElse` 与 `orElseGet` 方法的不同之处在于，无论 `Optional` 中是否存在值，`orElse` 总会创建新字符串；而 `orElseGet` 使用 `Supplier`，仅在 `Optional` 为空时才执行。

如果值只是一个简单的字符串，那么 `orElse` 与 `orElseGet` 方法的区别可以忽略不计；如果 `orElse` 方法的参数是一个复杂对象，那么传入 `Supplier` 的 `orElseGet` 方法能确保仅在需要时才创建对象。相关示例如例 6-8 所示。

例 6-8　在 `orElseGet` 方法中使用 `Supplier`

```
Optional<ComplexObject> val = values.stream.findFirst()

val.orElse(new ComplexObject());         ❶
val.orElseGet(() -> new ComplexObject()) ❷
```

❶ 总是创建新对象

❷ 仅在需要时才创建对象



采用 `Supplier` 作为方法参数是**延迟执行**（deferred execution）或**惰性执行**（lazy execution）的一种应用，从而可以仅在需要时才在 `Supplier` 上调用 `get` 方法。[3]

注 3：参见 Venkat Subramaniam 撰写的《Groovy 程序设计》一书，第 6 章详细讨论了这个问题。

在 Java 标准库中，`orElseGet` 方法的实现如例 6-9 所示。

例 6-9　`orElseGet` 方法的实现

```
public T orElseGet(Supplier<? extends T> other) {
    return value != null ? value : other.get();  ❶
}
```

❶ `value` 是 `Optional` 中 `T` 类型的最终特性

`orElseThrow` 方法同样传入 `Supplier`，该方法的签名如下：

```
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
```

因此，当 `Optional` 包含值时，不会执行用作 `Supplier` 参数的构造函数引用，如例 6-10 所示。

例 6-10　采用 `orElseThrow` 作为 `Supplier`

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

System.out.println(first.orElseThrow(NoSuchElementException::new));
```

此外，`ifPresent` 方法支持提供一个仅当 `Optional` 包含值时才执行的 `Consumer`，如例 6-11 所示。

例 6-11　`ifPresent` 方法的应用

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

first.ifPresent(val -> System.out.println("Found an even-length string"));

first = Stream.of("five", "even", "length", "string", "values")
    .filter(s -> s.length() % 2 != 0)
    .findFirst();

first.ifPresent(val -> System.out.println("Found an odd-length string"));
```

执行上述程序将仅打印“Found an even-length string”消息。

另见

有关 `Supplier` 接口的讨论请参见范例 2.2，有关构造函数引用的讨论请参见范例 1.3，有关返回 `Optional` 的 `findAny` 和 `findFirst` 方法请参见范例 3.9。

6.3　getter和setter方法中的`Optional`

问题

用户希望在访问器（accessor）和更改器（mutator）中使用 `Optional`。

方案

在 Optional 中包装 getter 方法的结果，但不要对 setter 方法，尤其是特性（attribute）执行同样的操作。

讨论

在 Optional 数据类型与用户通信的过程中，操作结果可以合法为 null 而不会抛出 NullPointerException。但是，Optional 类被有意设计成**不可序列化**（non-serializable），所以不应使用它来包装类中的字段。

因此，在 getter 和 setter 方法中添加 Optional 的首选方案，是当 getter 返回时在其中包装可空特性（nullable attribute），但不要对 setter 执行同样的操作。相关示例如例 6-12 所示。

例 6-12　在 DAO（数据访问对象）层中使用 Optional

```
public class Department {
    private Manager boss;

    public Optional<Manager> getBoss() {
        return Optional.ofNullable(boss);
    }

    public void setBoss(Manager boss) {
        this.boss = boss;
    }
}
```

在本例中，boss 是 Department 类中的 Manager 特性，可以将其视为可空类型[4]。用户或许试图创建类型为 Optional<Manager> 的特性，但由于 Optional 不可序列化，Department 也不可序列化。

本例不要求用户包装 Optional 中的值以调用 setter 方法，但这是 setBoss 方法传入 Optional<Manager> 作为参数所必需的。Optional 用于表示一个可能合法为 null 的值，且客户端已经了解该值是否为 null，但内部实现并不关心该值是否为 null。

此外，在 getter 方法中返回 Optional<Manager> 将告知调用程序，此时 Department 可能有（也可能没有）boss。

本例的不足之处在于，多年以来，"JavaBeans" 规范基于特性"对称"地定义了 getter 和 setter。实际上，Java 将**属性**（property）——而不仅仅是特性（attribute）——定义为遵循标准模式的 getter 和 setter。而本范例讨论的方案违反了这种模式，getter 和 setter 不再是对称的。

有鉴于此，部分开发人员认为 getter 和 setter 方法中不应出现 Optional，它属于不应暴露给客户端的内部实现细节。

不过，本范例讨论的方案被使用 Hibernate 等 ORM（对象关系映射）工具的开源开发者

注 4：或许这是个一厢情愿但颇具吸引力的想法。

所广泛接受。首要问题是告知客户端，存在一个支持特定字段的可空数据库列（nullable database column），而不必强制客户端在 setter 方法中包装引用。

这似乎是一种合理的折中方案，但正如这些开发者所言，解决方案应根据具体情况而定。

另见

有关 `Optional.map` 方法的讨论请参见范例 6.5，有关在 `Optional` 中包装值的讨论请参见范例 6.1。

6.4 Optional.flatMap与Optional.map方法

问题

用户希望避免将一个 `Optional` 包装在另一个 `Optional` 中。

方案

使用 `Optional` 类定义的 `flatMap` 方法。

讨论

有关 `Stream` 接口定义的 `map` 和 `flatMap` 方法请参见范例 3.11。不过 `flatMap` 是一个通用的概念，同样可以用于 `Optional`。

`Optional.flatMap` 方法的签名如下：

```
<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)
```

`Optional.flatMap` 方法的签名与 `Optional.map` 方法类似，二者的 `Function` 参数应用于每个元素并生成一个结果，返回类型为 `Optional<U>`。具体而言，如果参数 `T` 存在，`Optional.flatMap` 方法将函数应用于 `T` 并返回 `Optional`，它包装包含的值；如果 `T` 不存在，方法将返回一个空 `Optional`。

根据范例 6.3 的讨论，数据访问对象（DAO）通常采用返回 `Optional` 的 getter 方法编写（如果属性可以为 `null`），但 setter 方法不会将参数包装在 `Optional` 中。例 6-13 显示了两个类，一个是 `Manager` 类，它包含非空字符串 `name`；另一个是 `Department` 类，它包含可空 `Manager` 特性 boss。

例 6-13　包含 `Optional` 的 DAO 层（部分）

```
public class Manager {
    private String name;                    ❶

    public Manager(String name) {
        this.name = name;
    }
```

```
        public String getName() {
            return name;
        }
    }

    public class Department {
        private Manager boss;                   ❷

        public Optional<Manager> getBoss() {
            return Optional.ofNullable(boss);
        }

        public void setBoss(Manager boss) {
            this.boss = boss;
        }
    }
```

❶ 假定不为 `null`，因此不需要 `Optional`

❷ 可能为 `null`，因此在 `Optional` 中包装 getter 方法并返回，但不要对 setter 方法执行同样的操作

如果客户端调用 `Department` 的 `getBoss` 方法，结果将被包装在 `Optional` 中，如例 6-14 所示。

例 6-14　返回 `Optional`

```
Manager mrSlate = new Manager("Mr. Slate");

Department d = new Department();
d.setBoss(mrSlate);                          ❶
System.out.println("Boss: " + d.getBoss());  ❷


Department d1 = new Department();             ❸
System.out.println("Boss: " + d1.getBoss()); ❹
```

❶ `Department` 中存在非空 `Manager`

❷ 打印 `Boss: Optional[Manager{name='Mr. Slate'}]`

❸ `Department` 中不存在 `Manager`

❹ 打印 `Boss: Optional.empty`

截至目前一切顺利。如果 `Department` 中存在 `Manager`，getter 方法将其包装在 `Optional` 中并返回；如果 `Department` 中不存在 `Manager`，getter 方法将返回一个空 `Optional`。

问题在于，无法通过在 `Optional` 上调用 `getName` 方法来获取 `Manager` 的 `name`。我们要么从 `Optional` 中取出包含的值，要么使用 `map` 方法（例 6-15）。

例 6-15　从 `Optional` 的 `Manager` 中提取 `name`

```
System.out.println("Name: " +
        d.getBoss().orElse(new Manager("Unknown")).getName());   ❶

System.out.println("Name: " +
        d1.getBoss().orElse(new Manager("Unknown")).getName());
```

```
System.out.println("Name: " + d.getBoss().map(Manager::getName));  ❷
System.out.println("Name: " + d1.getBoss().map(Manager::getName));
```

❶ 在调用 getName 方法之前从 Optional 中提取 boss

❷ 利用 map 方法将 getName 应用于包含的 Manager

仅当 map 方法所调用的 Optional 不为空时，该方法才会应用给定函数，因此这种方案更为简单。map 方法的详细讨论请参见范例 6.5。

不过，如果多个 Optional 链接在一起，情况将变得较为复杂。例 6-16 定义了一个名为 Company 的类，它只包含一个 Department（为简单起见，只考虑一个 Department 的情况）。

例 6-16　只包含一个 Department 的 Company

```
public class Company {
    private Department department;

    public Optional<Department> getDepartment() {
        return Optional.ofNullable(department);
    }

    public void setDepartment(Department department) {
        this.department = department;
    }
}
```

如果在 Company 类上调用 getDepartment 方法，结果将被包装在 Optional 中。为获取 Manager 的信息，似乎可以采用例 6-15 讨论的 map 方法，但这会导致一个 Optional 被包装在另一个 Optional 中，如例 6-17 所示。

例 6-17　包装在另一个 Optional 中的 Optional

```
Company co = new Company();
co.setDepartment(d);

System.out.println("Company Dept: " + co.getDepartment());         ❶

System.out.println("Company Dept Manager: " + co.getDepartment()
    .map(Department::getBoss));                                    ❷
```

❶ 打印 Company Dept: Optional[Department{boss=Manager{name='Mr.Slate'}}]

❷ 打印 Company Dept Manager: Optional[Optional[Manager{name='Mr.Slate'}]]

为解决这个问题，不妨使用 Optional.flatMap 方法。flatMap 方法可以将结构“展平”，因此只会返回一个 Optional。我们仍然按之前的方式创建 Company 类，然后应用 flatMap 方法，如例 6-18 所示。

例 6-18　在 Company 上应用 flatMap

```
System.out.println(
    co.getDepartment()                 ❶
        .flatMap(Department::getBoss)  ❷
        .map(Manager::getName));       ❸
```

❶ `Optional<Department>`

❷ `Optional<Manager>`

❸ `Optional<String>`

接下来，将 `Company` 也包装在 `Optional` 中，如例 6-19 所示。

例 6-19　在 `Optional` 的 `Company` 上应用 `flatMap`

```
Optional<Company> company = Optional.of(co);

System.out.println(
    company                             ❶
        .flatMap(Company::getDepartment)  ❷
        .flatMap(Department::getBoss)     ❸
        .map(Manager::getName)            ❹
);
```

❶ `Optional<Company>`

❷ `Optional<Department>`

❸ `Optional<Manager>`

❹ `Optional<String>`

不难看到，我们甚至可以将 `Company` 包装在 `Optional` 中，然后重复执行 `flatMap` 操作就能获取任何所需的属性，最后通过 `map` 操作结束。

另见

有关在 `Optional` 中包装值的讨论请参见范例 6.1，有关 `Stream.flatMap` 方法的讨论请参见范例 3.11，有关在 DAO 层中应用 `Optional` 的讨论请参见范例 6.3，有关 `Optional.map` 方法的讨论请参见范例 6.5。

6.5　Optional的映射

问题

用户希望将函数应用到 `Optional` 实例的集合，但前提是 `Optional` 实例包含值。

方案

使用 `Optional` 类定义的 `map` 方法。

讨论

假设存在一个包含员工 ID 值的列表，我们希望检索相应的员工实例集合。如果 `findEmployeeById` 方法具有以下签名，则搜索所有员工后将返回一个 `Optional` 实例的集

合，其中某些实例可能为空。例 6-20 显示了如何筛掉为空的 Optional。

```
public Optional<Employee> findEmployeeById(int id)
```

例 6-20　根据 ID 查找 Employee（使用 Stream.map 方法）

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById)        ❶
        .filter(Optional::isPresent))       ❷
        .map(Optional::get)                 ❸
        .collect(Collectors.toList());
}
```

❶ Stream<Optional<Employee>>

❷ 删除空的 Optional

❸ 检索确定存在的值

如上所示，第一个 map 操作的结果是一个由 Optional 构成的流，每个 Optional 要么包含一名员工，要么为空。为提取包含的值，我们很自然会想到调用 get 方法。但必须注意，除非确定值存在，否则不要调用 get 方法。为避免出现错误，我们采用 filter 方法，它传入 Optional::isPresent 作为谓词以删除所有空 Optional，然后通过第二个 map 操作（传入 Optional::get）将各个 Optional 映射到它们所包含的值。

本例使用了 Stream 接口定义的 map 方法，而 Optional 类同样定义有 map 方法，其签名为：

```
<U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Optional.map 方法传入 Function 作为参数。如果 Optional 不为空，map 方法将提取包含的值并应用给定的函数，然后返回一个包含结果的 Optional；如果 Optional 为空，map 方法将返回一个空的 Optional。

我们使用 Optional.map 方法重写例 6-20 的查找操作，结果如例 6-21 所示。

例 6-21　根据 ID 查找 Employee（使用 Optional.map 方法）

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById)            ❶
        .flatMap(optional ->
            optional.map(Stream::of)            ❷
                    .orElseGet(Stream::empty))  ❸
        .collect(Collectors.toList());
}
```

❶ Stream<Optional<Employee>>

❷ 将非空 Optional<Employee> 转换为 Optional<Stream<Employee>>

❸ 从 Optional 中提取 Stream<Employee>

如果包含员工的 Optional 不为空，则在**包含的值**上调用 Stream::of，将其转换为该值的一个单元素流（one-element stream）并包装在 Optional 中，否则返回一个空的 Optional。

在本例中，如果根据某个员工 ID 找到了相应的员工，那么 findEmployeeById 方法将返回该值的 Optional<Employee>，optional.map(Stream::of) 方法随后返回一个 Optional，它包含存储该员工信息的单元素流，由此得到 Optional<Stream<Employee>>。接下来，orElseGet 方法将包含的值提取出来，生成 Stream<Employee>。

如果 findEmployeeById 方法返回为空的 Optional，optional.map(Stream::of) 方法同样将返回一个空的 Optional，而 orElseGet(Stream::empty) 方法将返回一个空的流。

由此得到 Stream<Employee> 元素与空流的组合，Stream.flatMap 方法的真正用途就在于此。仅对非空流而言，Stream.flatMap 方法将所有内容简化为 Stream<Employee>，因此 collect 方法可以将非空流作为员工列表返回。

相应的过程如图 6-1 所示。

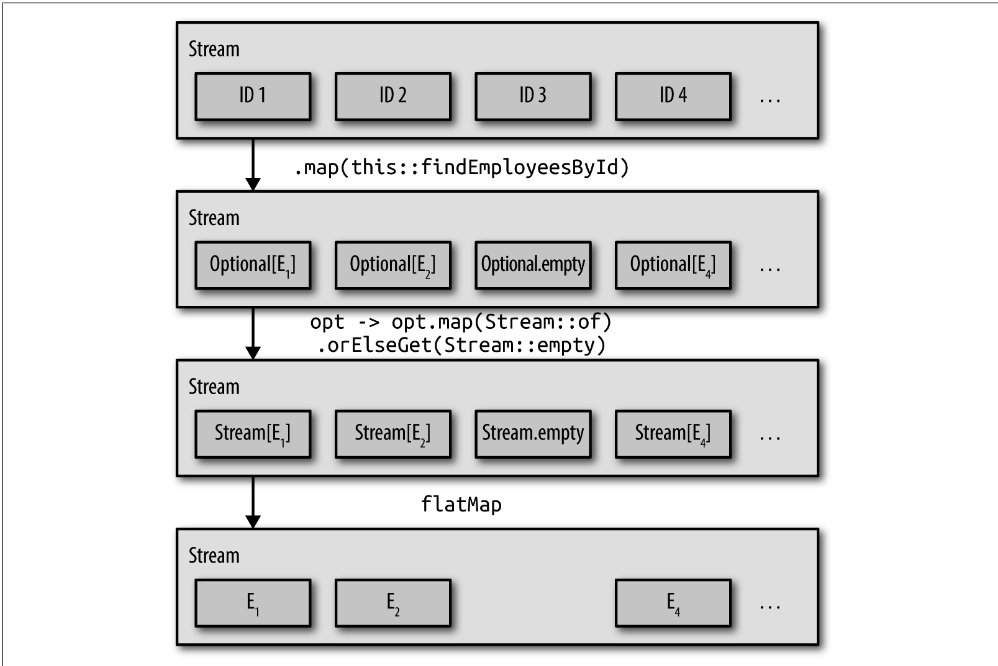


图 6-1：Optional.map 和 Optional.flatMap 操作

Optional.map 是一种或许有助于简化流处理代码的便捷[5]方法。对不熟悉 flatMap 操作的开发人员而言，之前讨论的 filter/map 方案显然更为直观，且得到的结果并无二致。

当然，我们可以在 Optional.map 方法中使用任何所需的函数。Javadoc 详细描述了如何将名称转换为文件输入流，其他应用请参见范例 6.4。

此外，Java 9 为 Optional 类引入了一个名为 stream 的新方法。如果 Optional 不为空，stream 方法将返回一个包装包含值的单元素流，否则返回一个空流。详见范例 10.6。

注 5：至少其设计思路如此。

另见

有关 `Optional` 在 DAO 层中的应用请参见范例 6.3，有关 `Stream.flatMap` 方法的讨论请参见范例 3.11，有关 `Optional.flatMap` 方法的讨论请参见范例 6.4，有关 Java 9 为 `Optional` 类引入的新方法请参见范例 10.6。

第 7 章

文件I/O

J2SE 1.4 引入了 NIO（non-blocking input/output，非阻塞输入 / 输出）包[1]，“N”有时也表示“新”（new）。而 Java 7 新增的 NIO.2 是对 NIO 的扩展，它定义了各种用于操作文件和目录的类。NIO.2 包括 `java.nio.file` 包，本章将对此讨论。Java 8 对其中的一些类（如 `java.nio.files.File`）进行强化，引入了若干用于处理流的方法。

遗憾的是，函数式编程所倡导的流式隐喻和输入 / 输出的同一术语相互冲突，这可能使用户感到困惑。例如，`java.nio.file.DirectoryStream` 接口与函数式流（functional stream）无关，该接口由使用传统 for-each 构造对目录树迭代的类来实现[2]。

这一章将重点讨论支持函数式流的 I/O 功能。Java 8 为 `java.nio.file.Files` 类引入了若干用于处理函数式流的新方法，这些方法如表 7-1 所示。请注意，`Files` 类中的所有方法均为静态方法。

表7-1：`java.nio.file.Files`类定义的返回流的方法

方法	返回类型
`lines`	`Stream<String>`
`list`	`Stream<Path>`
`walk`	`Stream<Path>`
`find`	`Stream<Path>`

这一章的范例将讨论上述方法。

注 1：大部分 Java 开发人员都惊讶于 NIO 的引入是如此之早。

注 2：更令人困惑的是，`DirectoryStream.Filter` 接口实际上属于函数式接口，尽管它也和函数式流无关。该接口仅用于批准目录树中选定的条目。

7.1 文件处理

问题

用户希望使用流来处理文本文件的内容。

方案

使用 `java.io.BufferedReader` 或 `java.nio.file.Files` 类定义的静态方法 `lines`，以流的形式返回文件内容。

讨论

在所有基于 FreeBSD 的 UNIX 系统（包括 macOS）中，/usr/share/dict/ 文件夹都包含《韦氏国际英语词典（第 2 版）》。web2 文件收录了大约 23 万个单词，每个单词在文件中占据一行。

假设我们希望查找词典中最长的 10 个单词。为此，我们首先使用 `Files.lines` 方法，将单词作为字符串流进行检索，然后执行 `map`、`filter` 等常规的流处理操作。相关示例如例 7-1 所示。

例 7-1 在 web2 文件中查找最长的 10 个单词

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2")) {
    lines.filter(s -> s.length() > 20)
        .sorted(Comparator.comparingInt(String::length).reversed())
        .limit(10)
        .forEach(w -> System.out.printf("%s (%d)%n", w, w.length()));
} catch (IOException e) {
    e.printStackTrace();
}
```

在本例中，`filter` 方法中的谓词筛掉了长度不足 20 个字符的单词，`sorted` 方法按长度对这些单词做降序排序，`limit` 方法在获取前 10 个单词后终止程序，然后打印这些单词。由于流是在 `try-with-resources` 代码块中打开的，当 `try` 代码块完成时，系统将自动关闭流与词典文件。

流与 `AutoCloseable` 接口

`Stream` 接口继承自 `BaseStream`，而它是 `AutoCloseable` 的子接口。因此，可以在 Java 7 新增的 `try-with-resources` 代码块中使用流。`try` 代码块执行完毕后，系统将自动调用 `close` 方法。它不仅会关闭流，还会调用流的流水线（stream pipeline）中的任何 `close` 处理程序以释放资源。

到目前为止，我们尚未接触 `try-with-resources` 包装器，因为之前讨论的流是从集合或在内存中生成的。而在本范例中，流是基于文件的，因此 `try-with-resources` 能确保词典文件也被关闭。

执行例 7-1 中的代码，结果如例 7-2 所示。

例 7-2　词典中最长的 10 个单词

```
formaldehydesulphoxylate (24)
pathologicopsychological (24)
scientificophilosophical (24)
tetraiodophenolphthalein (24)
thyroparathyroidectomize (24)
anthropomorphologically (23)
blepharosphincterectomy (23)
epididymodeferentectomy (23)
formaldehydesulphoxylic (23)
gastroenteroanastomosis (23)
```

如上所示，词典中有 5 个单词的长度为 24 个字符。结果之所以按字母顺序显示，只是因为原始文件中的单词是按字母顺序排序的。在例 7-1 中，如果为 `sorted` 方法的 `Comparator` 参数添加一个 `thenComparing` 子句，就能调整等长单词的排序方式了。

在 5 个长度为 24 个字符的单词之后，是 5 个长度为 23 个字符的单词，其中大部分单词来自医学领域。[3]

如果将 `Collectors.counting` 作为下游收集器，就能确定词典中每种长度的单词数量，如例 7-3 所示。

例 7-3　确定每种长度的单词数量（升序排序）

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2"))) {
    lines.filter(s -> s.length() > 20)
         .collect(Collectors.groupingBy(String::length, Collectors.counting()))
         .forEach((len, num) -> System.out.println(len + ": " + num));
}
```

上述代码使用收集器 `groupingBy` 创建一个 `Map`，其中键为单词长度，值为每种长度的单词数量。代码的执行结果如下：

```
21: 82
22: 41
23: 17
24: 5
```

上述输出虽然提供了部分信息，但并非特别有用。且结果按升序排序，这也可能不满足我们的要求。

另一种方案是采用 `Map.Entry` 接口新增的静态方法 `comparingByKey` 和 `comparingByValue`，二者均传入可选的 `Comparator`（相关讨论请参见范例 4.4）。如例 7-4 所示，通过比较器 `reverseOrder` 进行排序时，将返回自然顺序的相反顺序。

例 7-4　确定每种长度的单词数量（降序排序）

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2"))) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
```

注 3：好在 blepharosphincterectomy（眼轮匝肌切除术）与其字面意思无关，这是一个与减轻角膜上眼睑压力有关的单词。听起来很头疼？其实可能更头疼。

```
            .collect(Collectors.groupingBy(String::length, Collectors.counting()));

        map.entrySet().stream()
            .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
            .forEach(e -> System.out.printf("Length %d: %d words%n",
                e.getKey(), e.getValue()));
    }
```

程序的执行结果如下：

```
Length 24:  5 words
Length 23: 17 words
Length 22: 41 words
Length 21: 82 words
```

如果数据源不是文件，也可以使用 `BufferedReader` 类新增的 `lines` 方法（这种情况下，`lines` 是一个实例方法）。采用 `BufferedReader.lines` 方法对例 7-4 改写，结果如例 7-5 所示。

例 7-5　`BufferedReader.lines` 方法的应用

```
try (Stream<String> lines =
        new BufferedReader(
            new FileReader("/usr/share/dict/words")).lines()) {

    // 其余代码与例7-4相同
}
```

需要再次强调的是，由于 `Stream` 接口实现了 `AutoCloseable` 接口，当 `try-with-resources` 代码块关闭流时，底层 `BufferedReader` 也随之关闭。

另见

有关对映射排序的讨论请参见范例 4.4。

7.2　以流的形式检索文件

问题

用户希望将目录中的所有文件作为 `Stream` 进行处理。

方案

使用 `java.nio.file.Files` 类定义的静态方法 `list`。

讨论

`list` 方法传入 `Path` 作为参数，并返回一个包装 `DirectoryStream`[4] 的 `Stream`。由于 `DirectoryStream`

注 4：这是一个 IO 流而非函数式流。

接口继承自 AutoCloseable，try-with-resources 构造是使用 list 方法的最佳方式，如例 7-6 所示。

例 7-6　Files.list(path) 方法的应用

```
try (Stream<Path> list = Files.list(Paths.get("src/main/java"))) {
    list.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

如果在具有标准 Maven 或 Gradle 结构的项目根目录下执行上述代码，程序将打印 src/main/java 目录中所有文件和文件夹的名称。使用 try-with-resources 代码块，当 try 代码块执行完毕后，系统将在 Stream 上调用 close 方法，然后在底层 DirectoryStream 上调用 close 方法。请注意，目录和文件不是递归的。

执行本书配套的源代码（例 7-6），程序将输出目录和单个文件：

```
src/main/java/collectors
src/main/java/concurrency
src/main/java/datetime
...
src/main/java/Summarizing.java
src/main/java/tasks
src/main/java/UseFilenameFilter.java
```

list 方法的签名如下，其返回类型为 Stream<Path>，参数为目录的路径：

```
public static Stream<Path> list(Path dir) throws IOException
```

请注意，对非目录资源执行 list 方法将抛出 NotDirectoryException。

Javadoc 指出，list 方法返回的流具备**弱一致性**（weak consistency）。换言之，“流是线程安全的，但在迭代时不会冻结目录，因此它可能会（也可能不会）反映 list 方法返回后所发生的目录更新”。

另见

有关采用深度优先搜索遍历文件系统的讨论请参见范例 7.3。

7.3　文件系统的遍历

问题

用户希望对文件系统进行深度优先遍历（depth-first traversal）。

方案

使用 java.nio.file.Files 类定义的静态方法 walk。

讨论

walk 方法的签名如下：

```
public static Stream<Path> walk(Path start,
                                FileVisitOption... options)
                          throws IOException
```

walk 方法的参数为起始 Path 以及 FileVisitOption 值的可变参数列表。从起始路径开始对文件系统执行深度优先遍历，返回一个由 Path 实例惰性填充的 Stream。

由于返回的 Stream 封装了 DirectoryStream，仍然建议在 try-with-resources 代码块中使用 walk 方法，如例 7-7 所示。

例 7-7　文件树的遍历

```
try (Stream<Path> paths = Files.walk(Paths.get("src/main/java"))) {
    paths.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

walk 方法传入零个或多个 FileVisitOption 值作为第二个参数以及后续参数，不过本例并未使用任何 FileVisitOption 值。FileVisitOption 是 Java 1.7 引入的一种枚举类型，所包含的唯一枚举常量为 FOLLOW_LINKS。至少从理论上说，FOLLOW_LINKS 意味着文件树中可能存在循环，因此流将跟踪所访问的文件。一旦检测到循环，程序将抛出 FileSystemLoopException。

执行本书配套的源代码（例 7-7），输出类似于：

```
src/main/java
src/main/java/collectors
src/main/java/collectors/Actor.java
src/main/java/collectors/AddCollectionToMap.java
src/main/java/collectors/Book.java
src/main/java/collectors/CollectorsDemo.java
src/main/java/collectors/ImmutableCollections.java
src/main/java/collectors/Movie.java
src/main/java/collectors/MysteryMen.java
src/main/java/concurrency
src/main/java/concurrency/CommonPoolSize.java
src/main/java/concurrency/CompletableFutureDemos.java
src/main/java/concurrency/FutureDemo.java
src/main/java/concurrency/ParallelDemo.java
src/main/java/concurrency/SequentialToParallel.java
src/main/java/concurrency/Timer.java
src/main/java/datetime
...
```

程序采用惰性方式遍历路径，所生成的流必然包含至少一个元素（起始参数）。对于遇到的每条路径，程序将判断它是否为目录，是则遍历其中的所有条目，然后移动到下一个同级元素（sibling）。其结果是一种深度优先遍历。程序访问每个目录包含的所有条目之后，将关闭该目录。

walk 方法还包括以下重载形式：

```
public static Stream<Path> walk(Path start,
                                int maxDepth,
                                FileVisitOption... options)
                         throws IOException
```

其中，参数 maxDepth 是要访问的目录级别的最大值，0 表示只访问起始文件。如果不使用 maxDepth，可以通过 Integer.MAX_VALUE 值来指定应访问所有级别。

另见

有关如何列出一个目录中的所有文件请参见范例 7.2，有关文件搜索的讨论请参见范例 7.4。

7.4　文件系统的搜索

问题

用户希望查找文件树中满足给定属性的文件。

方案

使用 java.nio.file.Files 类定义的静态方法 find。

讨论

find 方法的签名如下：

```
public static Stream<Path> find(Path start,
                                int maxDepth,
                                BiPredicate<Path, BasicFileAttributes> matcher,
                                FileVisitOption... options)
                         throws IOException
```

可以看到，find 方法的签名与 walk 方法类似，但增加了一个用于决定是否应返回特定 Path 的匹配器 BiPredicate。find 方法从给定路径开始执行深度优先搜索（depth-first search），直至达到 maxDepth 指定的目录级别。对于每条路径，find 方法都会调用 BiPredicate 进行评估。如果指定为 FileVisitOption 枚举的值，则执行后面的链接。

BiPredicate 需要根据每个路径元素及其关联的 BasicFileAttributes 对象返回布尔值。如例 7-8 所示，程序将返回 fileio 包（参见本书配套源代码）中所有非目录文件的路径。

例 7-8　查找 fileio 包中的非目录文件

```
try (Stream<Path> paths =
    Files.find(Paths.get("src/main/java"), Integer.MAX_VALUE,
        (path, attributes) ->
            !attributes.isDirectory() && path.toString().contains("fileio"))) {
    paths.forEach(System.out::println);
} catch (IOException e) {
```

```
        e.printStackTrace();
    }
```

输出结果如下：

```
src/main/java/fileio/FileList.java
src/main/java/fileio/ProcessDictionary.java
src/main/java/fileio/SearchForFiles.java
src/main/java/fileio/WalkTheTree.java
```

对于遍历文件树时遇到的每一个文件，`find` 方法都会根据给定的 `BiPredicate` 进行评估，类似于在 `walk` 方法返回的 `Stream` 上调用筛选器。不过，Javadoc 认为 `find` 方法避免了对 `BasicFileAttributes` 对象的冗余检索，因而能提高程序的效率。

类似地，由于返回的 `Stream` 封装了 `DirectoryStream`，在关闭流的同时，底层数据源也随之关闭。有鉴于此，在 `try-with-resources` 代码块中使用 `find` 方法是首选方案。

另见

有关文件系统的遍历请参见范例 7.3。

第8章

java.time包

将 `java.util.Date` 类束之高阁才是正确之道。

——Tim Yates

在 Java 面世之初，标准库就引入了两种用于处理日期和时间的类，它们是 `java.util.Date` 和 `java.util.Calendar`，而前者堪称类**糟糕**设计的典范。浏览 API 可以发现，从 Java 1.1（1997 年 2 月发布）开始，`Date` 类中的所有方法就已被弃用。Java 1.1 推荐采用 `Calendar` 类处理日期和时间，但这个类同样存在不少问题。

在 Java 引入 `java.util.Date` 和 `java.util.Calendar` 类之前，枚举类型（enum）尚未出现，所以两种类在字段（如月份）中使用整型常量。两种类都是可变的，因而不是线程安全（thread safe）的。为处理实际开发中遇到的问题，标准库随后引入 `java.sql.Date` 作为 `java.util.Date` 的子类，但仍然没能彻底解决问题。

最终，Java SE 8 引入 `java.time` 包，这个全新的包从根本上解决了长久以来存在的诸多弊端。`java.time` 包基于 Joda-Time 库构建，它是一种免费的开源解决方案，多年来一直作为处理 Java 日期和时间的事实标准。实际上，Joda-Time 库的设计团队也参与了 `java.time` 包的开发，并建议开发人员在今后的工作中使用它。

`java.time` 包的开发遵循 JSR 310 规范（Date-Time API），并支持 ISO 8601 标准，且对闰年和个别地区实行的夏时制规则做了相应的调整。

这一章的范例将展示 `java.time` 包的各种应用，希望有助于解决读者可能遇到的某些基本问题，并在需要时提供进一步的信息。

感兴趣的读者可以阅读 Java 官方教程（Java Tutorials）中有关 Date-Time API 的介绍，它提供了系统而详尽的信息。

8.1 Date-Time API中的基本类

问题

用户希望使用 java.time 包引入的类来处理日期和时间。

方案

使用 Instant、Duration、Period、LocalDate、LocalTime、LocalDateTime、ZonedDateTime 等类定义的工厂方法。

讨论

Date-Time API 中的所有类均生成不可变实例，它们是线程安全的。由于这些类不提供公共构造函数（public constructor），需要采用工厂方法加以实例化。

读者应对 now 和 of 这两种静态工厂方法予以特别注意。now 方法根据当前日期或时间创建实例，示例代码如例 8-1 所示。

例 8-1　工厂方法 now

```
System.out.println("Instant.now():       " + Instant.now());
System.out.println("LocalDate.now():     " + LocalDate.now());
System.out.println("LocalTime.now():     " + LocalTime.now());
System.out.println("LocalDateTime.now(): " + LocalDateTime.now());
System.out.println("ZonedDateTime.now(): " + ZonedDateTime.now());
```

上述代码的输出如例 8-2 所示。

例 8-2　调用 now 方法的结果

```
Instant.now():       2017-06-20T17:27:08.184Z
LocalDate.now():     2017-06-20
LocalTime.now():     13:27:08.318
LocalDateTime.now(): 2017-06-20T13:27:08.319
ZonedDateTime.now(): 2017-06-20T13:27:08.319-04:00[America/New_York]
```

如例 8-2 所示，所有输出值均使用 ISO 8601 标准格式。日期的基本格式为 yyyy-MM-dd，而时间的基本格式为 hh:mm:ss.sss。LocalDateTime 类将两种格式合二为一，中间用大写字母 T 隔开。ZonedDateTime 类用于显示包含时区信息的日期和时间，其后添加了一个 UTC 偏移量（UTC offset）以及一个**地区名**（region name），本例分别为 -04:00 和 America/New_York。此外，Instant 类定义的 toString 方法将输出以祖鲁时间（Zulu time）[1] 显示的当前时间（精确到纳秒）。

类似地，Year、YearMonth 与 MonthDay 类也定义了 now 方法。

注 1：即 UTC 时间，因为北约音标字母（NATO phonetic alphabet）采用“Zulu”表示“Z”，而“Z”在 UTC 中表示零时区。例如，“03:06 UTC”可以表示为“03:06Z”（“03:06”和“Z”之间没有空格）。——译者注

静态工厂方法 of 用于生成新的值。对 LocalDate 类而言，of 方法的参数为年、月（枚举或整型）、日。



所有 of 方法的月份字段经过重载，以接受 Month 枚举（如 Month.JANUARY）或起始值为 1 的整数。不过，由于 java.util.Calendar 类定义的整型常量从 0 开始（即 Calendar.JANUARY 为 0），需注意避免出现差一错误（off-by-one error）。如有可能，应尽量使用 Month 枚举。

LocalTime 类定义的 of 方法包括多种重载形式，根据可用的小时、分、秒以及纳秒值获取当前日期。LocalDateTime 类定义的 of 方法同样包括多种重载形式，根据可用的年、月、日、小时、分、秒以及纳秒值获取当前日期和时间。相关应用如例 8-3 所示。

例 8-3　of 方法在日期 / 时间类中的应用

```
System.out.println("First landing on the Moon:");
LocalDate moonLandingDate = LocalDate.of(1969, Month.JULY, 20);
LocalTime moonLandingTime = LocalTime.of(20, 18);
System.out.println("Date: " + moonLandingDate);
System.out.println("Time: " + moonLandingTime);

System.out.println("Neil Armstrong steps onto the surface: ");
LocalTime walkTime = LocalTime.of(20, 2, 56, 150_000_000);
LocalDateTime walk = LocalDateTime.of(moonLandingDate, walkTime);
System.out.println(walk);
```

输出如下：

```
First landing on the Moon:
Date: 1969-07-20
Time: 20:18
Neil Armstrong steps onto the surface:
1969-07-20T20:02:56.150
```

LocalTime.of 方法的最后一个参数是纳秒。本例在数值中使用下划线以增强可读性，这是 Java 7 引入的一个特性。

Instant 类对时间轴上的单一瞬时点（single instantaneous point）建模，可以用于记录应用程序中的事件时间戳。

ZonedDateTime 类将日期和时间与通过 ZoneId 类获取的时区信息结合在一起，时区以 UTC 偏移量的形式表示。

时区 ID 包括两种类型。

- 相对于 UTC/ 格林尼治标准时间的固定偏移量（fixed offset），如 -05:00。
- 地理区域（geographical region），如 America/Chicago。

严格来说，还存在第三种时区 ID，即相对于祖鲁时间的偏移量，它由 Z 和相应的数值构成。

java.time.zone.ZoneRules 类定义了调整偏移量的规则，这些规则通过 java.time.zone.ZoneRulesProvider 类载入。ZoneRules 类包括 isDaylightSavings(Instant) 等方法。

静态方法 systemDefault 用于获取系统默认时区（ZoneId 的当前值），而可用时区 ID 的完整列表由静态方法 getAvailableZoneIds 提供：

```
Set<String> regionNames = ZoneId.getAvailableZoneIds();
System.out.println("There are " + regionNames.size() + " region names");
```

以 jdk1.8.0_131 为例，它包括 600 个地区名[2]。

Date-Time API 使用方法名的标准前缀。如果读者熟悉表 8-1 列出的前缀，不难猜出相应方法的用途。[3]

表8-1：Date-Time API中各种方法所用的前缀

方法	类型	用途
of	静态工厂	创建实例
from	静态工厂	将输入参数转换为目标类（target class）的实例
parse	静态工厂	解析输入字符串
format	实例	生成格式化输出
get	实例	返回对象状态的一部分
is	实例	查询对象状态
with	实例	通过修改现有对象的某个元素来创建新对象
plus，minus	实例	分别通过现有对象的增减来创建新对象
to	实例	将对象转换为另一种类型
at	实例	将对象与另一个对象合并

前面已讨论过 of 方法；有关 parse 和 format 方法的讨论请参见范例 8.5；有关 with 方法的讨论请参见范例 8.2，它是 set 方法的不可变等效形式；有关 plus 和 minus 方法的应用请参见范例 8.2。

例 8-4 显示了利用 at 方法为本地日期和时间添加时区。

例 8-4　为 LocalDateTime 添加时区信息

```
LocalDateTime dateTime = LocalDateTime.of(2017, Month.JULY, 4, 13, 20, 10);
ZonedDateTime nyc = dateTime.atZone(ZoneId.of("America/New_York"));
System.out.println(nyc);

ZonedDateTime london = nyc.withZoneSameInstant(ZoneId.of("Europe/London"));
System.out.println(london);
```

输出结果如下：

```
2017-07-04T13:20:10-04:00[America/New_York]
2017-07-04T18:20:10+01:00[Europe/London]
```

在本例中，withZoneSameInstant 方法传入一个 ZonedDateTime，并查找另一个时区的日期和时间。

注 2：或许不只作者感觉地区名的数量是如此之多。

注 3：根据 Java 官方教程提供的常用前缀表格编写。

java.time 包引入了 Month 和 DayOfWeek 两种枚举。Month 包括标准日历中 12 个月份的常量（从 JANUARY 到 DECEMBER），也定义了许多便利的方法，如例 8-5 所示。

例 8-5　Month 枚举定义的部分方法

```
System.out.println("Days in Feb in a leap year: " +
    Month.FEBRUARY.length(true));          ❶
System.out.println("Day of year for first day of Aug (leap year): " +
    Month.AUGUST.firstDayOfYear(true));  ❶
System.out.println("Month.of(1): " + Month.of(1));
System.out.println("Adding two months: " + Month.JANUARY.plus(2));
System.out.println("Subtracting a month: " + Month.MARCH.minus(1));
```

❶ 参数为 boolean leapYear

输出如下：

```
Days in Feb in a leap year: 29
Day of year for first day of Aug (leap year): 214
Month.of(1): JANUARY
Adding two months: MARCH
Subtracting a month: FEBRUARY
```

在本例中，最后两条语句分别使用 plus 和 minus 方法创建了新的实例。



java.time 包中的类是不可变的，如果实例方法（如 plus、minus 或 with）试图修改某个类，将生成一个新的实例。

DayOfWeek 枚举包括表示 7 个工作日的常量（从 MONDAY 到 SUNDAY）。所有工作日的 int 值均遵循 ISO 标准，因此 MONDAY 为 1，SUNDAY 为 7。

另见

有关解析和格式化的讨论请参见范例 8.5，有关将现有日期和时间转换为新的日期和时间请参见范例 8.2，有关 Duration 和 Period 类的应用请参见范例 8.8。

8.2　根据现有实例创建日期和时间

问题

用户希望修改 Date-Time API 中某个类的现有实例。

方案

如果需要进行简单的增减操作，使用 plus 或 minus 方法；对于其他操作，使用 with 方法。

讨论

Date-Time API 中的所有实例都是不可变的。一旦创建 LocalDate、LocalTime、LocalDateTime 或 ZonedDateTime，就无法修改它们。这对保持线程安全而言十分有利，不过如何根据现有实例创建新的实例呢？

以 LocalDate 类为例，它定义了多种对日期进行增减操作的方法，包括：

- LocalDate plusDays(long daysToAdd)
- LocalDate plusWeeks(long weeksToAdd)
- LocalDate plusMonths(long monthsToAdd)
- LocalDate plusYears(long yearsToAdd)

上述方法均返回一个新的 LocalDate，它是当前日期的副本，并添加了指定的值。

LocalTime 类也定义了类似的方法：

- LocalTime plusNanos(long nanosToAdd)
- LocalTime plusSeconds(long secondsToAdd)
- LocalTime plusMinutes(long minutesToAdd)
- LocalTime plusHours(long hoursToAdd)

类似地，每种方法均返回一个新的 LocalTime，它是当前时间的副本，并添加了指定的值。此外，LocalDateTime 类囊括了 LocalDate 和 LocalTime 类中用于处理日期和时间增减的所有方法。例 8-6 显示了各种 plus 方法在 LocalDate 和 LocalTime 类中的应用。

例 8-6　plus 方法在 LocalDate 和 LocalTime 类中的应用

```
@Test
public void localDatePlus() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    LocalDate start = LocalDate.of(2017, Month.FEBRUARY, 2);

    LocalDate end = start.plusDays(3);
    assertEquals("2017-02-05", end.format(formatter));

    end = start.plusWeeks(5);
    assertEquals("2017-03-09", end.format(formatter));

    end = start.plusMonths(7);
    assertEquals("2017-09-02", end.format(formatter));

    end = start.plusYears(2);
    assertEquals("2019-02-02", end.format(formatter));
}

@Test
public void localTimePlus() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_TIME;

    LocalTime start = LocalTime.of(11, 30, 0, 0);
    LocalTime end = start.plusNanos(1_000_000);
    assertEquals("11:30:00.001", end.format(formatter));
```

```
        end = start.plusSeconds(20);
        assertEquals("11:30:20", end.format(formatter));

        end = start.plusMinutes(45);
        assertEquals("12:15:00", end.format(formatter));

        end = start.plusHours(5);
        assertEquals("16:30:00", end.format(formatter));
    }
```

不少类还包括其他两种形式的 `plus` 和 `minus` 方法。以 `LocalDateTime` 类为例，`plus` 和 `minus` 方法的签名如下：

```
LocalDateTime plus(long amountToAdd, TemporalUnit unit)
LocalDateTime plus(TemporalAmount amountToAdd)

LocalDateTime minus(long amountToSubtract, TemporalUnit unit)
LocalDateTime minus(TemporalAmount amountToSubtract)
```

对于 `LocalDate` 和 `LocalDate` 类，`plus` 和 `minus` 方法的格式与 `LocalDateTime` 类相同，具有相应的返回类型。有趣的是，不妨将 `minus` 方法视为具有否定形式的 `plus` 方法。

对传入 `TemporalAmount` 的方法而言，参数通常为 `Period` 或 `Duration`，但也可以是任何实现 `TemporalAmount` 接口的类型。该接口定义了 `addTo` 和 `subtractFrom` 两种方法：

```
Temporal addTo(Temporal temporal)
Temporal subtractFrom(Temporal temporal)
```

跟踪调用栈（call stack）可以看到，调用 `minus` 委托给带有否定参数的 `plus`，而 `plus` 委托给 `TemporalAmount.addTo(Temporal)`，`TemporalAmount.addTo(Temporal)` 再回调 `plus(long, TemporalUnit)`，它将执行实际的操作[4]。

例 8-7 显示了 `plus` 和 `minus` 方法的相关应用。

例 8-7　`plus` 和 `minus` 方法的应用

```
@Test
public void plus_minus() throws Exception {
    Period period = Period.of(2, 3, 4); // 两年3个月零4天
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    LocalDateTime end = start.plus(period);

    assertEquals("2019-05-06T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.plus(3, ChronoUnit.HALF_DAYS);
    assertEquals("2017-02-03T23:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.minus(period);
    assertEquals("2014-10-29T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

注 4：老天，多么间接的操作！

```
    end = start.minus(2, ChronoUnit.CENTURIES);
    assertEquals("1817-02-02T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.plus(3, ChronoUnit.MILLENNIA);
    assertEquals("5017-02-02T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}
```



当 API 调用 `TemporalUnit` 时，提供的实现类为 `ChronoUnit`，它定义了许多方便的枚举常量（enum constant）可供使用。

此外，每种类都定义了一系列 `with` 方法，可以一次修改一个字段。

`with` 方法用于处理常用的日期和时间，某些方法颇为有趣。以 `LocalDateTime` 类为例：

```
LocalDateTime withNano(int nanoOfSecond)
LocalDateTime withSecond(int second)
LocalDateTime withMinute(int minute)
LocalDateTime withHour(int hour)
LocalDateTime withDayOfMonth(int dayOfMonth)
LocalDateTime withDayOfYear(int dayOfYear)
LocalDateTime withMonth(int month)
LocalDateTime withYear(int year)
```

例 8-8 显示了各种 `with` 方法的应用。

例 8-8　`with` 方法在 `LocalDateTime` 类中的应用

```
@Test
public void with() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    LocalDateTime end = start.withMinute(45);
    assertEquals("2017-02-02T11:45:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withHour(16);
    assertEquals("2017-02-02T16:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withDayOfMonth(28);
    assertEquals("2017-02-28T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withDayOfYear(300);
    assertEquals("2017-10-27T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withYear(2020);
    assertEquals("2020-02-02T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}
```

```
@Test(expected = DateTimeException.class)
public void withInvalidDate() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    start.withDayOfMonth(29);
}
```

由于 2017 年并非闰年，无法将日期设置为 2 月 29 日，第二个测试将抛出 `DateTimeException`。

`with` 方法也可以传入 `TemporalAdjuster` 或 `TemporalField`：

```
LocalDateTime with(TemporalAdjuster adjuster)
LocalDateTime with(TemporalField field, long newValue)
```

传入 `TemporalField` 的 `with` 方法允许字段解析日期以使其有效。如例 8-9 所示，程序传入 1 月的最后一天，并尝试将月份改为 2 月（“2 月 31 日”）。此时，根据 Javadoc 的描述，系统将选择前一个有效日期，即 2 月的最后一天（2 月 28 日）。

例 8-9　月份调整（无效）

```
@Test
public void temporalField() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.JANUARY, 31, 11, 30);
    LocalDateTime end = start.with(ChronoField.MONTH_OF_YEAR, 2);
    assertEquals("2017-02-28T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}
```

可想而知，日期和时间的处理涉及一些相当复杂的规则，不过 Javadoc 对此做了系统而详尽的描述。

范例 8.3 将讨论传入 `TemporalAdjuster` 的 `with` 方法。

另见

有关 `TemporalAdjuster` 和 `TemporalQuery` 的讨论请参见范例 8.3。

8.3　调节器与查询

问题

给定一个时态值（temporal value），用户希望根据自定义逻辑对其进行调整，或检索给定值的相关信息。

方案

创建 `TemporalAdjuster` 或规划 `TemporalQuery` 接口。

讨论

`TemporalAdjuster` 和 `TemporalQuery` 接口中的类不仅提供使用 Date-Time API 中各种类的有

趣方式，也提供有用的内置方法以实现用户自定义的方法。本范例将对此做讨论。

1. TemporalAdjuster的应用

TemporalAdjuster 接口定义了一个名为 adjustInto 的方法，它传入 Temporal 值作为参数，并返回调整后的值。而 TemporalAdjusters 类包括一系列用作静态方法的调节器（adjuster），或许能为开发带来一定便利。

以 LocalDateTime 类为例，我们可以通过时态对象（temporal object）上的 with 方法使用 TemporalAdjuster：

```
LocalDateTime with(TemporalAdjuster adjuster)
```

虽然也可以使用 TemporalAdjuster 接口定义的 adjustInto 方法，但 with 方法应作为首选。

我们首先讨论 TemporalAdjusters 类，它定义了多种便利的方法：

```
static TemporalAdjuster firstDayOfNextMonth()
static TemporalAdjuster firstDayOfNextYear()
static TemporalAdjuster firstDayOfYear()

static TemporalAdjuster firstInMonth(DayOfWeek dayOfWeek)
static TemporalAdjuster lastDayOfMonth()
static TemporalAdjuster lastDayOfYear()
static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)

static TemporalAdjuster next(DayOfWeek dayOfWeek)
static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)
static TemporalAdjuster previous(DayOfWeek dayOfWeek)
static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)
```

例 8-10 的用例显示了上述方法在实际开发中的应用。

例 8-10　TemporalAdjusters 类定义的部分静态方法

```
@Test
public void adjusters() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    LocalDateTime end = start.with(TemporalAdjusters.firstDayOfNextMonth());
    assertEquals("2017-03-01T11:30", end.toString());

    end = start.with(TemporalAdjusters.next(DayOfWeek.THURSDAY));
    assertEquals("2017-02-09T11:30", end.toString());

    end = start.with(TemporalAdjusters.previousOrSame(DayOfWeek.THURSDAY));
    assertEquals("2017-02-02T11:30", end.toString());
}
```

有趣之处在于编写自定义调节器。TemporalAdjuster 是一个函数式接口，所包含的单一抽象方法为：

```
Temporal adjustInto(Temporal temporal)
```

在讨论时态调节器（temporal adjuster）时，Java 官方教程以 PaydayAdjuster 类为例演示了自定义调节器的应用：假设员工在一个月中领取两次工资，且发薪日是每月 15 日和最后

一天；如果某个发薪日为周末，则提前到周五。

为便于参考，例 8-11 完整复制了这个示例的代码。请注意，adjustInto 方法已被添加到实现 TemporalAdjuster 接口的 PaydayAdjuster 类中。

例 8-11　PaydayAdjuster 类（取自 Java 官方教程）

```java
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.Temporal;
import java.time.temporal.TemporalAdjuster;
import java.time.temporal.TemporalAdjusters;

public class PaydayAdjuster implements TemporalAdjuster {
    public Temporal adjustInto(Temporal input) {
        LocalDate date = LocalDate.from(input);  ❶
        int day;
        if (date.getDayOfMonth() < 15) {
            day = 15;
        } else {
            day = date.with(TemporalAdjusters.lastDayOfMonth())
                      .getDayOfMonth();
        }
        date = date.withDayOfMonth(day);
        if (date.getDayOfWeek() == DayOfWeek.SATURDAY ||
                date.getDayOfWeek() == DayOfWeek.SUNDAY) {
            date = date.with(TemporalAdjusters.previous(DayOfWeek.FRIDAY));
        }

        return input.with(date);
    }
}
```

❶ from 方法可以将任何时态对象转换为 LocalDate

以 2017 年 7 月为例运行程序，其中 7 月 15 日是周六，7 月 31 日是周一。例 8-12 的测试显示，调节器可以正确处理 2017 年 7 月的发薪日。

例 8-12　测试调节器

```java
@Test
public void payDay() throws Exception {
  TemporalAdjuster adjuster = new PaydayAdjuster();
  IntStream.rangeClosed(1, 14)
           .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
           .forEach(date ->
               assertEquals(14, date.with(adjuster).getDayOfMonth()));

  IntStream.rangeClosed(15, 31)
           .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
           .forEach(date ->
               assertEquals(31, date.with(adjuster).getDayOfMonth()));
}
```

虽然上述程序可以运行，但仍然存在改进的空间。首先，在 Java 8 之前，如果不使用其他机制就无法创建日期流（例如，本例需要计算天数）。这种情况在 Java 9 中得以改变。Java

9 新增了一种返回日期流的方法 datesUntil，详细讨论请参见范例 10.7。

其次，为实现 TemporalAdjuster 接口，程序创建了 PaydayAdjuster 类。由于 TemporalAdjuster 属于函数式接口，不妨改为提供 lambda 表达式或方法引用作为实现。

如例 8-13 所示，我们创建一个名为 Adjusters 的工具类，它包括进行各种操作所需的静态方法。

例 8-13　工具类 Adjusters

```
public class Adjusters {                                    ❶
    public static Temporal adjustInto(Temporal input) {  ❷
        LocalDate date = LocalDate.from(input);
        // 与例8-11的实现相同
        return input.with(date);
    }
}
```

❶ 不实现 TemporalAdjuster 接口

❷ 静态方法无须实例化

重写后的测试如例 8-14 所示。

例 8-14　测试调节器（使用方法引用）

```
@Test
public void payDayWithMethodRef() throws Exception {
    IntStream.rangeClosed(1, 14)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
              assertEquals(14,
                  date.with(Adjusters::adjustInto).getDayOfMonth())); ❶

    IntStream.rangeClosed(15, 31)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
              assertEquals(31,
                  date.with(Adjusters::adjustInto).getDayOfMonth()));
}
```

❶ adjustInto 的方法引用

如果存在多个时态调节器，上述方案可能更为通用。

2. TemporalQuery的应用

TemporalQuery 接口用作时态对象中 query 方法的参数。以 LocalDate 类为例，query 方法的签名如下：

```
<R> R query(TemporalQuery<R> query)
```

query 方法调用 TemporalQuery.queryFrom(TemporalAccessor) 方法（传入 this 作为参数），并返回所需的查询。TemporalAccessor 接口定义的所有方法均可用于查询操作。

Date-Time API 还包括一个名为 TemporalQueries 的类，它定义了许多常见查询的常量：

```
static TemporalQuery<Chronology>   chronology()
static TemporalQuery<LocalDate>    localDate()
static TemporalQuery<LocalTime>    localTime()
static TemporalQuery<ZoneOffset>   offset()
static TemporalQuery<TemporalUnit> precision()
static TemporalQuery<ZoneId>       zone()
static TemporalQuery<ZoneId>       zoneId()
```

例 8-15 的简单测试展示了部分方法的应用。

例 8-15　TemporalQueries 类定义的部分方法

```
@Test
public void queries() throws Exception {
    assertEquals(ChronoUnit.DAYS,
        LocalDate.now().query(TemporalQueries.precision()));
    assertEquals(ChronoUnit.NANOS,
        LocalTime.now().query(TemporalQueries.precision()));
    assertEquals(ZoneId.systemDefault(),
        ZonedDateTime.now().query(TemporalQueries.zone()));
    assertEquals(ZoneId.systemDefault(),
        ZonedDateTime.now().query(TemporalQueries.zoneId()));
}
```

与 `TemporalAdjuster` 接口类似，有趣之处在于编写自定义查询。`TemporalQuery` 接口包含的单一抽象方法为：

```
R queryFrom(TemporalAccessor temporal)
```

如果给定参数 `TemporalAccessor`，我们可以编写一个名为 `daysUntilPirateDay` 的方法，以计算指定日期与国际海盗模仿日（International Talk Like A Pirate Day，9 月 19 日）[5] 之间的天数，如例 8-16 所示。

例 8-16　计算指定日期与国际海盗模仿日之间的天数

```
private long daysUntilPirateDay(TemporalAccessor temporal) {
    int day = temporal.get(ChronoField.DAY_OF_MONTH);
    int month = temporal.get(ChronoField.MONTH_OF_YEAR);
    int year = temporal.get(ChronoField.YEAR);
    LocalDate date = LocalDate.of(year, month, day);
    LocalDate tlapd = LocalDate.of(year, Month.SEPTEMBER, 19);
    if (date.isAfter(tlapd)) {
        tlapd = tlapd.plusYears(1);
    }

    return ChronoUnit.DAYS.between(date, tlapd);
}
```

由于 `daysUntilPirateDay` 方法的签名与 `TemporalQuery` 接口包含的单一抽象方法 `queryFrom` 相互兼容，可以通过方法引用来调用，如例 8-17 所示。

注 5：例如，“喂，伙计！我打算把你加入我的 LinkedIn 社交网络。”

例 8-17　通过方法引用使用 TemporalQuery

```
@Test
public void pirateDay() throws Exception {
    IntStream.range(10, 19)
             .mapToObj(n -> LocalDate.of(2017, Month.SEPTEMBER, n))
             .forEach(date ->
                assertTrue(date.query(this::daysUntilPirateDay) <= 9));
    IntStream.rangeClosed(20, 30)
             .mapToObj(n -> LocalDate.of(2017, Month.SEPTEMBER, n))
             .forEach(date -> {
                Long days = date.query(this::daysUntilPirateDay);
                assertTrue(days >= 354 && days < 365);
            });
}
```

上述方案也可用于创建自定义查询。

8.4　将java.util.Date转换为java.time.LocalDate

问题

用户希望将 java.util.Date 或 java.util.Calendar 类转换为 java.time 包中相应的类。

方案

转换时既可以利用 Instant 类作为中介，也可以使用 java.sql.Date 和 java.sql.Timestamp 类提供的方法，还可以使用字符串或整数。

讨论

新的 java.time 包并未提供太多的内置方式来转换 java.util 包中用于处理标准日期和时间的类，这点或许会让读者感到讶异。

为了将 java.util.Date 类转换为 java.time.LocalDate 类，一种方案是调用 toInstant 方法来创建 Instant，然后应用系统默认时区（ZoneId），并从生成的 ZonedDateTime 中提取出 LocalDate，如例 8-18 所示。

例 8-18　利用 Instant 作为中介，将 java.util.Date 类转换为 java.time.LocalDate 类

```
public LocalDate convertFromUtilDateUsingInstant(Date date) {
    return date.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
}
```

java.util.Date 类包含日期和时间信息，但并不提供时区信息[6]，因此它相当于 java.time.Instant 类。将 atZone 方法应用到系统默认时区将创建 ZonedDateTime，之后就能从中提取出 LocalDate。

注 6：打印 java.util.Date 时，字符串采用 Java 默认的时区进行格式化。

此外，借由 java.sql.Date 和 java.sql.Timestamp 类定义的一些方法，也可以方便地将 java.util.Date 类转换为 java.time.LocalDate 类，相关示例请参见例 8-19 和例 8-20。

例 8-19　java.sql.Date 类中的转换方法

```
LocalDate   toLocalDate()
static Date valueOf(LocalDate date)
```

例 8-20　java.sql.Timestamp 类中的转换方法

```
LocalDateTime    toLocalDateTime()
static Timestamp valueOf(LocalDateTime dateTime)
```

如例 8-21 所示，我们创建一个名为 ConvertDate 的类，从而方便地实现转换。

例 8-21　将 java.util 包中的类转换为 java.time 包中相应的类

```
package datetime;

import java.sql.Timestamp;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Date;

public class ConvertDate {
    public LocalDate convertFromSqlDatetoLD(java.sql.Date sqlDate) {
        return sqlDate.toLocalDate();
    }

    public java.sql.Date convertToSqlDateFromLD(LocalDate localDate) {
        return java.sql.Date.valueOf(localDate);
    }

    public LocalDateTime convertFromTimestampToLDT(Timestamp timestamp) {
        return timestamp.toLocalDateTime();
    }

    public Timestamp convertToTimestampFromLDT(LocalDateTime localDateTime) {
        return Timestamp.valueOf(localDateTime);
    }
}
```

既然所需的方法基于 java.sql.Date 类，那么如何转换 java.util.Date（大部分开发人员仍在使用）以及 java.sql.Date 类呢？一种方案是利用 java.sql.Date 类提供的构造函数，根据给定的毫秒时间值创建一个 Date 对象（long 型数据）。

纪元时间与 Java

在基于 Unix 的操作系统中，Unix **纪元时间**（Unix epoch）也称为 Unix 时间戳（Unix timestamp）或 POSIX 时间（POSIX time），定义为从 1970 年 1 月 1 日 00:00:00 UTC 起经过的秒数，不考虑闰秒。目前，计算机的系统时钟均以纪元时间为基础。

需要注意的是，由于 Unix 纪元时间采用 32 位有符号整数存储经过的秒数，将在 2038 年 1 月 19 日 03:14:07 UTC 时溢出。在这一刻之后，全球所有 32 位操作系统的时间将突然

跳回 1901 年 12 月 13 日。这就是所谓的“2038 年问题”（Year 2038 Problem）[7]。尽管到那个时候，所有操作系统应该都已升级为 64 位，但可能仍有部分嵌入式系统尚未更新[8]。

Java 采用毫秒作为经过时间的单位，这或许会让情况变得更加糟糕。不过使用 `long` 而非 `int` 存储经过时间，可以将溢出问题推后数千年。

`java.util.Date` 类定义了一个返回 `long` 型数据的 `getTime` 方法，而 `java.sql.Date` 类定义了一个传入该 `long` 值作为参数的构造函数 `setTime`[9]。

因此，借由 `java.sql.Date` 类，也可以将 `java.util.Date` 实例转换为 `java.time.LocalDate` 实例，如例 8-22 所示。

例 8-22　将 `java.util.Date` 类转换为 `java.time.LocalDate` 类

```
public LocalDate convertUtilDateToLocalDate(java.util.Date date) {
    return new java.sql.Date(date.getTime()).toLocalDate();
}
```

实际上，早在 Java 1.1 发布时，整个 `java.util.Date` 类就已被弃用，并推荐采用 `java.util.Calendar` 类作为替代。`Calendar` 实例与 `java.time` 包中相应实例之间的转换可以通过 `toInstant` 方法完成，并根据时区进行调整，如例 8-23 所示。

例 8-23　将 `java.util.Calendar` 类转换为 `java.time. ZonedDateTime` 类

```
public ZonedDateTime convertFromCalendar(Calendar cal) {
    return ZonedDateTime.ofInstant(cal.toInstant(), cal.getTimeZone().toZoneId());
}
```

上述方法使用 `ZonedDateTime` 类。`LocalDateTime` 类也定义了一个名为 `ofInstant` 的方法，不过由于某种原因，该方法传入 `ZoneId` 作为第二个参数。因为 `LocalDateTime` 类并不包含时区信息，这显得颇为奇怪。有鉴于此，改用 `ZonedDateTime` 类定义的 `ofInstant` 方法或许更加直观。

如果完全不必考虑时区信息，也可以在 `Calendar` 类上显式地使用各种 getter 方法，直接转换为相应的 `LocalDateTime`，如例 8-24 所示。

例 8-24　利用 getter 方法将 `java.util.Calendar` 转换为 `java.time.LocalDateTime`

```
public LocalDateTime convertFromCalendarUsingGetters(Calendar cal) {
    return LocalDateTime.of(cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH),
        cal.get(Calendar.DAY_OF_MONTH),
        cal.get(Calendar.HOUR),
        cal.get(Calendar.MINUTE),
```

注 7：参见维基百科的详细介绍。（32 位有符号整数的最大值为 0x7FFFFFFF，即 2^31 − 1= 2147483647 秒，也就是 2038 年 1 月 19 日 03:14:07 UTC。可以通过 Epoch Converter 方便地将 Unix 纪元时间转换为人类可读的格式。——译者注）

注 8：作者届时想必已安全退休，不过当 2038 年问题发生时，希望作者使用的呼吸机不会受到影响。

注 9：事实上，`setTime` 是 `java.sql.Date` 类中唯一一个未被弃用的构造函数，可以利用该方法来设置现有的 `Date` 对象。

```
        cal.get(Calendar.SECOND));
}
```

另一种方案是根据 Calendar 类生成一个经过格式化的字符串，然后将其解析为 LocalDateTime，如例 8-25 所示。

例 8-25　生成并解析时间戳字符串

```
public LocalDateTime convertFromUtilDateToLDUsingString(Date date) {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
    return LocalDateTime.parse(df.format(date),
        DateTimeFormatter.ISO_LOCAL_DATE_TIME);
}
```

上述方案并非特别理想，不过了解它并无坏处。此外，Calendar 类并未提供能直接转换为 ZonedDateTime 的方法，但 GregorianCalendar 类定义的 toZonedDateTime 方法可以实现这种转换，如例 8-26 所示。

例 8-26　将 java.util.GregorianCalendar 类转换为 java.time.ZonedDateTime 类

```
public ZonedDateTime convertFromGregorianCalendar(Calendar cal) {
    return ((GregorianCalendar) cal).toZonedDateTime();
}
```

上述程序可以执行，不过前提是采用公历（Gregorian calendar）。由于 GregorianCalendar 类是 Calendar 类的唯一实现，这种前提应该成立，但无法百分之百确定。

最后，Java 9 为 LocalDate 类引入了 ofInstant 方法，使得转换操作更为简单，如例 8-27 所示。

例 8-27　将 java.util.Date 类转换为 java.time.LocalDate 类（仅针对 Java 9）

```
public LocalDate convertFromUtilDateJava9(Date date) {
    return LocalDate.ofInstant(date.toInstant(), ZoneId.systemDefault());
}
```

这种方案更直接，但仅能在 Java 9 中使用。

8.5　解析与格式化

问题

用户希望解析或格式化 Date-Time API 中的类。

方案

DateTimeFormatter 类用于创建日期 / 时间格式，可以在解析和格式化中使用。

讨论

DateTimeFormatter 类提供大量预定义格式化器（predefined formatter），包括常量（如 ISO_LOCAL_DATE）、模式字母（如 uuuu-MMM-dd）以及本地化样式（如 ofLocalizedDate(dateStyle)）。

好在解析和格式化的过程并不复杂。在 Date-Time API 中，所有主要的类均提供 parse 和 format 方法。以 LocalDate 类为例，其 parse 和 format 方法的签名如例 8-28 所示。

例 8-28　LocalDate 类定义的 parse 和 format 方法

```
static LocalDate parse(CharSequence text)  ❶
static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
       String    format(DateTimeFormatter formatter)
```

❶ 使用 ISO_LOCAL_DATE

解析和格式化的应用如例 8-29 所示。

例 8-29　对 LocalDateTime 解析和格式化

```
LocalDateTime now = LocalDateTime.now();
String text = now.format(DateTimeFormatter.ISO_DATE_TIME);  ❶
LocalDateTime dateTime = LocalDateTime.parse(text);         ❷
```

❶ 将 LocalDateTime 格式化为字符串

❷ 将字符串解析为 LocalDateTime

因此，我们可以调整日期 / 时间格式、区域设置等各种参数。部分应用如例 8-30 所示。

例 8-30　对日期进行格式化

```
LocalDate date = LocalDate.of(2017, Month.MARCH, 13);

System.out.println("Full : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));
System.out.println("Long : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)));
System.out.println("Medium : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)));
System.out.println("Short : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));

System.out.println("France : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(Locale.FRANCE)));
System.out.println("India : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(new Locale("hin", "IN"))));
System.out.println("Brazil : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(new Locale("pt", "BR"))));
System.out.println("Japan : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(Locale.JAPAN)));

Locale loc = new Locale.Builder()
        .setLanguage("sr")
        .setScript("Latn")
        .setRegion("RS")
        .build();
System.out.println("Serbian: " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(loc)));
```

执行上述程序，输出类似于：[10]

```
Full   : Monday, March 13, 2017
Long   : March 13, 2017
Medium : Mar 13, 2017
Short  : 3/13/17

France : lundi 13 mars 2017
India  : Monday, March 13, 2017
Brazil : Segunda-feira, 13 de Março de 2017
Japan  : 2017 年3 月13 日
Serbian: ponedeljak, 13. mart 2017
```

由于 parse 和 format 方法分别抛出 DateTimeParseException 和 DateTimeException，用户可能希望在自己的代码中捕获它们。

我们也可以通过 DateTimeFormatter 类提供的 ofPattern 方法创建自定义格式化器，Javadoc 详细描述了所有可以使用的合法值。ofPattern 方法的应用如例 8-31 所示。

例 8-31　自定义格式化模式

```
ZonedDateTime moonLanding = ZonedDateTime.of(
        LocalDate.of(1969, Month.JULY, 20),
        LocalTime.of(20, 18),
        ZoneId.of("UTC")
);
System.out.println(moonLanding.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));

DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("uuuu/MMMM/dd hh:mm:ss a zzz GG");
System.out.println(moonLanding.format(formatter));

formatter = DateTimeFormatter.ofPattern("uuuu/MMMM/dd hh:mm:ss a VV xxxxx");
System.out.println(moonLanding.format(formatter));
```

输出如下：

```
1969-07-20T20:18:00Z[UTC]
1969/July/20 08:18:00 PM UTC AD
1969/July/20 08:18:00 PM UTC +00:00
```

有关 DateTimeFormatter 类的用途以及各种模式字母的含义，请参见 Javadoc。不过读者无须担心，格式化的过程并不复杂。

接下来，我们通过夏时制（daylight savings time）问题来展示本地化日期 / 时间格式化器的应用。北美东部时区（EST）从 2018 年 3 月 11 日凌晨 2 时起实行夏时制，将时钟调快一小时。那么在 3 月 11 日凌晨 2 时 30 分时，某个地区的日期和时间是多少呢？相关示例如例 8-32 所示。

注 10：有传言说，作者有意选择稀有语种和输出格式，只是为了测试 O'Reilly Media 能否正确打印出相应的结果。不过至少就读者所知，这并非事实。

例 8-32　将时钟调快一小时

```
ZonedDateTime zdt = ZonedDateTime.of(2018, 3, 11, 2, 30, 0, 0,
    ZoneId.of("America/New_York"));
System.out.println(
    zdt.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL)));
```

在本例中，`ZonedDateTime.of` 方法传入年、月、日、小时、分、秒、纳秒以及时区作为参数。除时区（`ZoneId`）外，其他字段均为 `int` 类型，即无法使用 `Month` 枚举。

输出如下：

```
Sunday, March 11, 2018 3:30:00 AM EDT
```

可以看到，程序将时间从凌晨 2 时 30 分调整为凌晨 3 时 30 分。

8.6　查找具有非整数小时偏移量的时区

问题

用户希望查找所有具有非整数小时偏移量（non-integral hour offset）的时区。

方案

获取每个时区的时区偏移量，并计算总秒数除以 3600 之后的剩余时间。

讨论

大部分时区的 UTC 偏移量为小时的整数。例如，通常所说的北美东部时区（EST）为 `UTC-05:00`，而欧洲中部时间（CET）为 `UTC+01:00`。不过也存在 UTC 偏移量为半小时甚至 45 分钟的时区，如印度标准时间（IST）为 `UTC+05:30`，而查塔姆标准时间（CHAST）为 `UTC+12:45`。本范例将讨论利用 `java.time` 包查找所有具有非整数小时偏移量的时区。

如例 8-33 所示，我们通过 `ZoneOffset` 类查找每个时区 ID 相对于 UTC 的偏移量，并将其总秒数与 3600 秒（1 小时）进行比较。

例 8-33　查找每个时区 ID 的偏移量（以秒为单位）

```
public class FunnyOffsets {
    public static void main(String[] args) {
        Instant instant = Instant.now();
        ZonedDateTime current = instant.atZone(ZoneId.systemDefault());
        System.out.printf("Current time is %s%n%n", current);

        System.out.printf("%10s %20s %13s%n", "Offset", "ZoneId", "Time");
        ZoneId.getAvailableZoneIds().stream()
                .map(ZoneId::of)  ❶
                .filter(zoneId -> {
                    ZoneOffset offset = instant.atZone(zoneId).getOffset(); ❷
                    return offset.getTotalSeconds() % (60 * 60) != 0;  ❸
                })
```

```
                .sorted(comparingInt(zoneId ->
                        instant.atZone(zoneId).getOffset().getTotalSeconds()))
                .forEach(zoneId -> {
                    ZonedDateTime zdt = current.withZoneSameInstant(zoneId);
                    System.out.printf("%10s %25s %10s%n",
                        zdt.getOffset(), zoneId,
                        zdt.format(DateTimeFormatter.ofLocalizedTime(
                            FormatStyle.SHORT)));
                });
    }
}
```

❶ 将地区 ID（字符串）映射到时区 ID

❷ 计算偏移量

❸ 仅返回偏移量无法被 3600 整除的时区 ID

`ZoneId.getAvailableZoneIds` 静态方法返回 `Set<String>`，表示所有可用的时区 ID；`ZoneId.of` 方法将生成的字符串流转换为 `ZoneId` 实例流。

在本例中，筛选器中的 lambda 表达式首先将 `atZone` 方法应用到 `Instant` 以创建 `ZonedDateTime`，然后应用 `getOffset` 方法。最后，利用 `ZoneOffset` 类定义的 `getTotalSeconds` 方法获取时区偏移量（以秒为单位）。根据 Javadoc 的描述，`getTotalSeconds` 方法是“访问偏移量的主要方式，它返回小时、分、秒字段的总和（以秒为单位），作为一个偏移量添加到给定的时间”。仅当总秒数无法被 3600（`60 sec/min * 60 min/hour`）整除时，筛选器中的 `Predicate` 才返回 `true`。

在打印结果前，程序对生成的 `ZoneId` 实例排序。`sorted` 方法传入 `Comparator` 作为参数。本例使用 `Comparator` 接口定义的静态方法 `comparingInt`，它生成一个根据给定整数键排序的 `Comparator`。程序同样采用 `getTotalSeconds` 方法获取时区偏移量（以秒为单位），`ZoneId` 实例根据偏移量进行排序。

接下来，针对每个 `ZoneId`，程序使用 `withZoneSameInstant` 方法计算默认时区中当前的 `ZonedDateTime`，以打印结果。打印的字符串将显示偏移量、时区 ID 以及相应时区中经过格式化的本地时间。

程序的执行结果如例 8-34 所示。

例 8-34　具有非整数小时偏移量的时区

```
Current time is 2016-08-08T23:12:44.264-04:00[America/New_York]

    Offset                ZoneId        Time
    -09:30      Pacific/Marquesas    5:42 PM
    -04:30        America/Caracas   10:42 PM
    -02:30       America/St_Johns   12:42 AM
    -02:30    Canada/Newfoundland   12:42 AM
    +04:30                   Iran    7:42 AM
    +04:30            Asia/Tehran    7:42 AM
    +04:30             Asia/Kabul    7:42 AM
    +05:30           Asia/Kolkata    8:42 AM
    +05:30           Asia/Colombo    8:42 AM
```

```
+05:30            Asia/Calcutta    8:42 AM
+05:45           Asia/Kathmandu    8:57 AM
+05:45            Asia/Katmandu    8:57 AM
+06:30             Asia/Rangoon    9:42 AM
+06:30             Indian/Cocos    9:42 AM
+08:45          Australia/Eucla   11:57 AM
+09:30          Australia/North   12:42 PM
+09:30     Australia/Yancowinna   12:42 PM
+09:30       Australia/Adelaide   12:42 PM
+09:30    Australia/Broken_Hill   12:42 PM
+09:30          Australia/South   12:42 PM
+09:30         Australia/Darwin   12:42 PM
+10:30      Australia/Lord_Howe    1:42 PM
+10:30            Australia/LHI    1:42 PM
+11:30          Pacific/Norfolk    2:42 PM
+12:45                  NZ-CHAT    3:57 PM
+12:45          Pacific/Chatham    3:57 PM
```

可以看到，将 `java.time` 包中的多个类结合在一起使用，就能解决复杂而有趣的问题。

8.7　根据UTC偏移量查找地区名

问题

给定某个 UTC 偏移量时，用户希望查找 ISO 8601 标准定义的地区名。

方案

根据给定的偏移量，筛选所有可用的时区 ID。

讨论

尽管“东部夏令时”（Eastern Daylight Time）和“印度标准时间”（Indian Standard Time）这样的时区名已广为人知，但它们并非 ISO 官方名称，其缩写“EDT”和“IST”在某些情况下甚至不是唯一的。ISO 8601 标准采用以下两种方式定义时区 ID。

- 根据地区名（region name），如“America/Chicago”。
- 根据以小时和分为单位的 UTC 偏移量（UTC offset），如“+05:30”。

那么，如何根据给定的 UTC 偏移量获取相应的地区名呢？对于任意给定的时间，虽然许多地区的 UTC 偏移量相同，但在给定偏移量的情况下，不难计算出相应的地区名列表。

`ZoneOffset` 类用于获取某个时区相对于格林尼治 /UTC 时间的偏移量。如果给定偏移量，也可以使用它筛选完整的地区名列表，如例 8-35 所示。

例 8-35　根据给定的偏移量，获取相应的地区名

```
public static List<String> getRegionNamesForOffset(ZoneOffset offset) {
    LocalDateTime now = LocalDateTime.now();
    return ZoneId.getAvailableZoneIds().stream()
```

```
            .map(ZoneId::of)
            .filter(zoneId -> now.atZone(zoneId).getOffset().equals(offset))
            .map(ZoneId::toString)
            .sorted()
            .collect(Collectors.toList());
}
```

在本例中，ZoneId.getAvailableZoneIds 方法返回一个由字符串构成的 List，每个字符串通过 ZoneId.of 方法映射到相应的 ZoneId。利用 LocalDateTime 类定义的 atZone 方法确定 ZoneId 对应的 ZonedDateTime 之后，就能得到所有 ZoneId 的 ZoneOffset，并筛掉其中不匹配的 ZoneOffset。接下来，程序将结果映射到字符串、对字符串排序并将它们收集到 List 中。

反过来，如何获取 ZoneOffset 呢？一种方案是利用给定的 ZoneId，如例 8-36 所示。

例 8-36　根据给定的时区，获取相应的偏移量

```
public static List<String> getRegionNamesForZoneId(ZoneId zoneId) {
    LocalDateTime now = LocalDateTime.now();
    ZonedDateTime zdt = now.atZone(zoneId);
    ZoneOffset offset = zdt.getOffset();

    return getRegionNamesForOffset(offset);
}
```

上述代码适用于任何给定的 ZoneId。

例 8-37 显示了如何获取与用户当前位置对应的地区名列表。

例 8-37　获取当前的地区名

```
@Test
public void getRegionNamesForSystemDefault() throws Exception {
    ZonedDateTime now = ZonedDateTime.now();
    ZoneId zoneId = now.getZone();
    List<String> names = getRegionNamesForZoneId(zoneId);

    assertTrue(names.contains(zoneId.getId()));
}
```

如果希望通过 GMT 偏移量（以小时和分为单位）获取地区名，也可以使用 ZoneOffset 类定义的 ofHoursMinutes 方法，如例 8-38 所示。

例 8-38　根据给定的偏移量（以小时和分为单位），获取地区名

```
public static List<String> getRegionNamesForOffset(int hours, int minutes) {
    ZoneOffset offset = ZoneOffset.ofHoursMinutes(hours, minutes);
    return getRegionNamesForOffset(offset);
}
```

如例 8-39 所示，我们编写几个测试，验证在给定偏移量的情况下，能否成功获取相应的地区名。

例 8-39　根据给定的偏移量，测试能否成功获取地区名

```
@Test
public void getRegionNamesForGMT() throws Exception {
    List<String> names = getRegionNamesForOffset(0, 0);
```

```
        assertTrue(names.contains("GMT"));
        assertTrue(names.contains("Etc/GMT"));
        assertTrue(names.contains("Etc/UTC"));
        assertTrue(names.contains("UTC"));
        assertTrue(names.contains("Etc/Zulu"));
    }

    @Test
    public void getRegionNamesForNepal() throws Exception {
        List<String> names = getRegionNamesForOffset(5, 45);

        assertTrue(names.contains("Asia/Kathmandu"));
        assertTrue(names.contains("Asia/Katmandu"));
    }

    @Test
    public void getRegionNamesForChicago() throws Exception {
        ZoneId chicago = ZoneId.of("America/Chicago");
        List<String> names = RegionIdsByOffset.getRegionNamesForZoneId(chicago);

        assertTrue(names.contains("America/Chicago"));
        assertTrue(names.contains("US/Central"));
        assertTrue(names.contains("Canada/Central"));
        assertTrue(names.contains("Etc/GMT+5") || names.contains("Etc/GMT+6"));
    }
```

有关时区列表的详细信息，请参见维基百科。

8.8　获取事件之间的时间

问题

用户希望获取两个事件之间的时间。

方案

如果需要将时间转换为人类可读的格式，使用时态类（temporal class）定义的 `between` 或 `until` 方法，或 `Period` 类定义的 `between` 方法以生成 `Period` 对象；如果不需要转换时间格式，则使用以秒和纳秒为单位对时间量进行建模的 `Duration` 类。

讨论

在 `java.time.temporal` 包中，`TemporalUnit` 接口由定义在同一个包中的 `ChronoUnit` 枚举实现。`TemporalUnit` 接口定义了一个名为 `between` 的方法，它传入两个 `TemporalUnit` 实例并返回二者之间的时间量（`long` 型数据）：

```
long between(Temporal temporal1Inclusive,
             Temporal temporal2Exclusive)
```

请注意，起始时间和终止时间的类型必须相互兼容。在计算时间量之前，实现将第二个类型转换为第一个类型的实例。如果终止时间早于起始时间，则结果为负。

between 方法的返回值是两个参数之间的完整“单位”数[11]，这为使用 ChronoUnit 定义的枚举常量提供了便利。

例如，假设我们希望获取当天与今后某个特定日期之间的天数。由于需要处理的是天数，使用 ChronoUnit 定义的枚举常量 ChronoUnit.DAYS，如例 8-40 所示。

例 8-40　计算当天到 2020 年美国选举日（11 月 3 日）之间的天数

```
LocalDate electionDay = LocalDate.of(2020, Month.NOVEMBER, 3);
LocalDate today = LocalDate.now();

System.out.printf("%d day(s) to go...%n",
    ChronoUnit.DAYS.between(today, electionDay));
```

程序在 ChronoUnit.DAYS 上调用 between 方法，并返回两个日期之间的天数。ChronoUnit 定义的其他枚举常量包括 HOURS、WEEKS、MONTHS、YEARS、DECADES、CENTURIES 等。[12]

1. java.time.Period类

我们可以利用 Period 类将时间分解为符合人类阅读习惯的年、月、日格式。不少基本类的 until 方法都返回 Period：

```
// java.time.LocalDate类
Period until(ChronoLocalDate endDateExclusive)
```

重写例 8-40，结果如例 8-41 所示。

例 8-41　通过 Period 类获取年、月、日

```
LocalDate electionDay = LocalDate.of(2020, Month.NOVEMBER, 3);
LocalDate today = LocalDate.now();

Period until = today.until(electionDay); ❶

years  = until.getYears();
months = until.getMonths();
days   = until.getDays();
System.out.printf("%d year(s), %d month(s), and %d day(s)%n",
        years, months, days);
```

❶ 相当于 Period.between(today, electionDay)

从本例的注释可以看到，Period 类同样定义了一个名为 between 的静态方法，其用法与 until 方法并无区别。选择哪种样式均可，但应以有助于提高代码可读性为原则。

简而言之，如果需要将时间转换为人类可读的格式（如年、月、日），应使用 Period 类。

注 11：返回值为整数（whole number）。例如，15:00 和 16:59 之间的小时数为 1 小时，即便它和 2 小时只差 1 分钟。——译者注

注 12：无论相信与否，ChronoUnit 中还包括一个名为 FOREVER 的枚举常量。如果读者在开发中确实用到了这个常量，请告诉作者，作者很想看看它的实际应用。（根据 Javadoc 的描述，FOREVER 是一个表示永恒概念的人造单位，通常与 TemporalField 一起使用，代表年份和时代这样的无界字段。——译者注）

2. java.time.Duration类

Duration 类表示以秒和纳秒为单位的时间量，通常与 Instant 一起使用，结果可以转换为许多其他类型。Duration 类存储一个表示秒的 long 型数据以及一个表示纳秒的 int 型数据。如果终点早于起点，则结果为负。

例 8-42 显示了如何利用 Duration 实现简单的计时。

例 8-42　对某个方法进行计时

```
public static double getTiming(Instant start, Instant end) {
    return Duration.between(start, end).toMillis() / 1000.0;
}

Instant start = Instant.now();
// 调用需要计时的方法
Instant end = Instant.now();
System.out.println(getTiming(start, end) + " seconds");
```

这种对方法计时的手段并不高明，但胜在简单易行。

Duration 类定义了 toDays、toHours、toMillis、toMinutes、toNanos 等多种转换方法，这就是本例中 getTiming 方法使用 toMillis 并除以 1000 的原因。

第 9 章

并行与并发

这一章将讨论 Java 8 中的并行（parallelization）和并发（concurrency）。某些概念可以追溯到早期版本（特别是 Java 5 引入的 `java.util.concurrent` 包），不过 Java 8 对并行和并发进行了强化，使得开发人员可以在更高的抽象层次上操作。

在讨论并行和并发时，有人会非常在意这两个术语之间的区别。我们对二者做一简单定义。

- **并发**：多个任务可以在重叠的时间段内运行。
- **并行**：多个任务可以同时运行。

并发设计指将一个任务分解为多个能同时运行的独立操作——即便目前尚未这样处理。换言之，并发应用程序由若干独立执行的进程组成。如果存在多个处理单元，就可以并行地实现这些并发任务，但性能是否会因此而提升将视情况而定[1]。

那么，并行为什么在某些情况下无助于性能提升呢？原因是多方面的。对 Java 而言，并行在默认情况下将任务分解成多个子任务，每个子任务被分配给通用 fork/join 线程池（common fork/join pool）并执行，最后将所有结果合并在一起。但是，所有操作都会引入开销，而很多预期的性能提升取决于问题映射到算法的程度。对于是否使用并行，不妨参考本章范例给出的指导方针。

Java 8 使并行变得简单易行。Clojure 语言之父 Rich Hickey 在 Strange Loop 2011 上以“Simple Made Easy”为题，对此做了精彩的阐述。Hickey 表示，一个基本概念是**简单**（simple）和**容易**（easy）这两个词具有不同的含义。简而言之，**简单**的事物在概念上并无歧义，而**容易**的事物在看似容易的背后或许隐藏了巨大的复杂性。例如，有些排序算法很

注 1：Go 语言之父 Rob Pike 在题为“Concurrency Is Not Parallelism”的演讲中，对这两个概念做了精彩且扼要的解释。感兴趣的读者可以观看上传到 YouTube 的演讲视频。

简单，有些则不然，但调用 `Stream.sorted` 方法总是很容易[2]。

并行和并发处理涉及多方面的内容，是一个令人颇感头疼的话题。Java 在面世之初就引入了支持多线程访问的底层机制，如 `Object` 类定义的 `wait`、`notify`、`notifyAll` 方法以及 `synchronized` 关键字。但使用这样的原语（primitives）实现并发难如登天，因此 Java 5 引入了 `java.util.concurrent` 包。借由 `ExecutorService`、`BlockingQueue` 接口以及 `ReentrantLock` 类，开发人员得以在更高的抽象层次上处理并发。尽管如此，并发管理仍然不是一项轻松的工作，特别是遇到堪称梦魇的"共享可变状态"（shared mutable state）时。

而在 Java 8 中，请求并行流不再是难事，因为只需进行一次方法调用。问题在于，性能提升并不简单。之前存在的所有问题其实仍然存在，它们只是被隐藏在表面之下而已。

并发和并行的讨论分散于全书，这一章的范例无法涵盖全部内容。[3] 本章旨在介绍各种可用的机制及其用法。掌握这些知识和概念后，读者可以将它们应用到开发中，并根据实际情况决定是否使用并发和并行。

9.1　将顺序流转换为并行流

问题

无论默认情况如何，用户希望创建顺序流（sequential stream）或并行流（parallel stream）。

方案

既可以使用 `Collection` 接口定义的 `stream` 或 `parallelStream` 方法，也可以使用 `BaseStream` 接口定义的 `sequential` 或 `parallel` 方法。

讨论

默认情况下，Java 创建的流都是顺序流。在 `BaseStream` 接口（`Stream` 的父接口）中，可以通过 `isParallel` 方法判断流是否采用并行方式执行。

从例 9-1 可以看到，所有采用标准机制创建的流都是顺序流。

例 9-1　创建顺序流（JUnit 测试的一部分）

```
@Test
public void sequentialStreamOf() throws Exception {
    assertFalse(Stream.of(3, 1, 4, 1, 5, 9).isParallel());
}
```

注 2：在《星际迷航：下一代》中，饰演皮卡德舰长（Captain Picard）的帕特里克 • 斯图尔特（Patrick Stewart）为"简单"与"容易"做了另一个很好的注脚：编剧试图向斯图尔特描述进入行星轨道所需的全部详细步骤，斯图尔特答道："何必这么繁琐？告诉我'标准轨道，少尉'就够了。"

注 3：感兴趣的读者可以阅读 Brian Goetz 撰写的 *Java Concurrency in Practice* 一书（由 Addison-Wesley Professional 于 2006 年 5 月出版），以及 Venkat Subramaniam 撰写的 *Programming Concurrency on the JVM* 一书（由 Pragmatic Bookshelf 于 2011 年 9 月出版）。

```
@Test
public void sequentialIterateStream() throws Exception {
    assertFalse(Stream.iterate(1, n -> n + 1).isParallel());
}

@Test
public void sequentialGenerateStream() throws Exception {
    assertFalse(Stream.generate(Math::random).isParallel());
}

@Test
public void sequentialCollectionStream() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertFalse(numbers.stream().isParallel());
}
```

如果数据源为集合，可以通过 `parallelStream` 方法返回一个可能的并行流，如例 9-2 所示。

例 9-2　`parallelStream` 方法的应用

```
@Test
public void parallelStreamMethodOnCollection() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertTrue(numbers.parallelStream().isParallel());
}
```

之所以强调"可能的"，是因为 `parallelStream` 方法在某些情况下也会返回顺序流，但默认返回的是并行流。Javadoc 指出，仅当创建自定义 `spliterator` 时才会返回顺序流，不过这种情况相当罕见。[4]

如例 9-3 所示，也可以通过在现有流上使用 `parallel` 方法来创建并行流。

例 9-3　在流上使用 `parallel` 方法

```
@Test
public void parallelMethodOnStream() throws Exception {
    assertTrue(Stream.of(3, 1, 4, 1, 5, 9)
            .parallel()
            .isParallel());
}
```

与 `parallel` 方法相对的是 `sequential` 方法，它将返回顺序流，如例 9-4 所示。

例 9-4　将并行流转换为顺序流

```
@Test
public void parallelStreamThenSequential() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertFalse(numbers.parallelStream()
            .sequential()
            .isParallel());
}
```

注 4：毋庸置疑，这是一个有趣的话题。不过受篇幅所限，本书不对此做讨论。

不过转换时需谨慎行事，否则可能落入陷阱。假设我们计划创建一个流水线（pipeline），其中一部分处理可以并行地完成，而其他处理仍然按顺序执行。那么，我们很容易写出如例 9-5 所示的代码。

例 9-5　并行流到顺序流的切换（与预期结果不同）

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
List<Integer> nums = numbers.parallelStream()  ❶
      .map(n -> n * 2)
      .peek(n -> System.out.printf("%s processing %d%n",
             Thread.currentThread().getName(), n))
      .sequential()                            ❷
      .sorted()
      .collect(Collectors.toList());
```

❶ 请求并行流

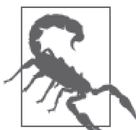
❷ 先切换为顺序流，再排序

上述代码的含义不难理解：先将所有数字倍增，再排序。由于倍增函数是无状态（stateless）且关联的（associative），采用并行操作可谓顺理成章。然而，排序本质上属于顺序操作[5]。

在本例中，`peek` 方法用于显示进行处理的线程的名称，程序在调用 `parallelStream` 方法之后、`sequential` 方法之前调用 `peek`。输出如下：

```
main processing 6
main processing 2
main processing 8
main processing 2
main processing 10
main processing 18
```

可以看到，`main` 线程完成了所有的处理。换言之，尽管调用了 `parallelStream` 方法，但返回的仍然是顺序流。这是什么原因呢？读者或许还记得，流在达到终止表达式（terminal expression）前不会进行任何操作，即在达到终止表达式时才会评估流的状态。在本例中，由于 `collect` 方法之前最后调用的是 `sequential` 方法，程序将返回顺序流，并对元素做相应处理。



流在执行时既可以是并行的，也可以是顺序执行的。`parallel` 或 `sequential` 方法能有效地设置或撤销设置一个布尔值，并在达到终止表达式时进行检查。

如果确有必要以并行方式处理部分流，而以顺序方式处理流的其他部分，建议使用两个单独的流。虽然这样处理也存在不少问题，但目前尚无更好的解决方案。

注 5：可以这样理解：使用并行流排序意味着将区间划分为若干相等的部分，然后分别对每部分排序，再尝试将所有经过排序的子区间合并在一起。那么从整体来看，最终的输出其实并没有实现排序。

9.2　并行流的优点

问题

用户希望了解并行流的优势所在。

方案

在合适的条件下应用并行流。

讨论

Stream API 能方便地将顺序流转换为并行流，不过性能提升与否需要视情况而定。请记住，切换到并行流是一种优化操作，但首先应保证代码可以正常工作，再决定是否有必要使用并行流。建议根据实际情况进行决策。

在 Java 8 中，并行流默认使用通用 fork/join 线程池来分发任务。线程池大小等于 JVM 可用的处理器数量，它由 `Runtime.getRuntime().availableProcessors()` 确定。[6] 无论是将任务分解为多个子任务，还是将所有子任务的结果合并为最终输出，都会为 fork/join 线程池的管理引入开销。

为了使这些额外的开销物有所值，应在满足以下要求时再使用并行流：

- 数据量较大
- 每个元素的处理较为耗时
- 数据源易于分解
- 操作是无状态且关联的

前两项要求经常被合二为一。如果 `N` 为数据元素的数量，`Q` 为每个元素所需的计算时间，则 `N` 与 `Q` 的乘积通常需要超过某个阈值，使用并行流才可能获得性能提升 [7]。根据第三项要求，应采用易于分解的数据结构（如数组）。最后，在并行处理时，如果操作是有状态或有序的，那么显然会出现问题。

为说明并行流对性能提升的效果，我们来看一个最简单的例子。如例 9-6 所示，程序为顺序流添加了为数不多的几个整数。

例 9-6　为顺序流添加整数

```
public static int doubleIt(int n) {
    try {
        Thread.sleep(100);              ❶
    } catch (InterruptedException ignore) {
    }
```

注 6：严格来说，线程池大小应为处理器数量减 1。但如果将主线程包括在内，则线程池大小与处理器数量相等。

注 7：一个约定俗成的公式是 `N * Q > 10,000`，不过似乎没有人会使用很大的 `Q` 值，因此很难解释为何将 10 000 作为阈值。

```
        return n * 2;
    }

    // 主线程
    Instant before = Instant.now();         ❷
    total = IntStream.of(3, 1, 4, 1, 5, 9)
            .map(ParallelDemo::doubleIt)
            .sum();
    Instant after = Instant.now();          ❷
    Duration duration = Duration.between(start, end);
    System.out.println("Total of doubles = " + total);
    System.out.println("time = " + duration.toMillis() + " ms");
```

❶ 人为引入延迟

❷ 获取倍增前后的时间

由于添加数字的过程极快，除非人为引入延迟，否则难以看出并行操作对性能提升的效果。在本例中，`N` 是如此之小（`N = 6`），因此通过引入 100 毫秒的延迟（`sleep(100)`）来扩展 `Q`。

默认情况下，Java 创建的流都是顺序流。由于每个元素的倍增操作都延迟 100 毫秒，处理 6 个元素所需的总时间约为 600 毫秒。程序的实际执行结果如下：

```
Total of doubles = 46
time = 621 ms
```

接下来，我们对代码进行微调，改为使用并行流。如例 9-7 所示，在 `map` 操作前插入 `parallel` 方法，其他代码保持不变。

例 9-7　使用并行流

```
total = IntStream.of(3, 1, 4, 1, 5, 9)
    .parallel()                 ❶
    .map(ParallelDemo::doubleIt)
    .sum();
```

❶ 使用并行流

在一台 8 核计算机上，实例化的 fork/join 线程池大小为 8[8]。换言之，流中的每个元素都可以被分配一个 CPU 核心（假设当前没有其他任务，稍后将讨论），因此所有倍增操作基本会同时进行。

再次执行程序，结果如下：

```
Total of doubles = 46
time = 112 ms
```

每个倍增操作延迟 100 毫秒，且有足够的线程对每个数字单独处理，因此整个计算仅需 100 多毫秒就能完成。

1. 利用JHM计时

众所周知，性能度量绝非易事，它与缓存、JVM 启动时间等多种因素有关。前面的示例只

注 8：线程池的实际大小为 7，但如果将主线程包括在内，则有 8 个独立的线程。

是粗略估算了顺序流和并行流的处理时间，如果希望获得更精确的测试结果，可以采用微基准测试框架（micro-benchmarking framework）。

JMH（Java Micro-benchmark Harness）是一种常用的 Java 微基准测试框架，它通过注解（annotation）来设置计时模式、范围、JVM 参数等。采用 JMH 重构前面的示例，结果如例 9-8 所示。

例 9-8　利用 JMH 对倍增操作计时

```
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
@Fork(value = 2, jvmArgs = {"-Xms4G", "-Xmx4G"})
public class DoublingDemo {
    public int doubleIt(int n) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException ignored) {
        }
        return n * 2;
    }

    @Benchmark
    public int doubleAndSumSequential() {
        return IntStream.of(3, 1, 4, 1, 5, 9)
                .map(this::doubleIt)
                .sum();
    }

    @Benchmark
    public int doubleAndSumParallel() {
        return IntStream.of(3, 1, 4, 1, 5, 9)
                .parallel()
                .map(this::doubleIt)
                .sum();
    }
}
```

根据默认设置，在一系列预热迭代（warmup iteration）[9] 后，JHM 将在两个独立的线程中执行 20 次迭代。典型的运行结果如下：

```
Benchmark                            Mode  Cnt    Score   Error  Units
DoublingDemo.doubleAndSumParallel    avgt   40  103.523 ± 0.247  ms/op
DoublingDemo.doubleAndSumSequential  avgt   40  620.242 ± 1.656  ms/op
```

可以看到，这些值与粗略估算的结果基本相同：顺序处理的平均耗时约为 620 毫秒，而并行处理的平均耗时约为 103 毫秒。换言之，只要处理器的数量足够多，并行操作就能为 6

注 9：旨在优化 JIT，让系统进入稳定状态，以便测试结果更接近实际情况。——译者注

个数字各自分配一个单独的线程，这比连续执行每个运算的速度要快 6 倍。

2. 对基本数据类型求和

在上一节的示例中，由于 N 过小，为体现并行流对性能提升的效果，我们人为引入延迟以扩展 Q。接下来，我们将对泛型流（generic stream）和基本类型流（primitive stream）的迭代操作、并行操作以及顺序操作进行比较，并通过较大的 N 值观察不同操作的性能。



本节的示例并不复杂，它根据经典教程 *Mordern Java in Action* (*Second Edition*)[10] 中一个类似的示例改编而成。

例 9-9 显示了采用迭代方式对循环中的数字求和。

例 9-9　采用迭代方式对循环中的数字求和

```
public long iterativeSum() {
    long result = 0;
    for (long i = 1L; i <= N; i++) {
        result += i;
    }
    return result;
}
```

例 9-10 分别采用顺序方式和并行方式对 `Stream<Long>` 求和。

例 9-10　对泛型流求和

```
public long sequentialStreamSum() {
    return Stream.iterate(1L, i -> i + 1)
            .limit(N)
            .reduce(0L, Long::sum);
}

public long parallelStreamSum() {
    return Stream.iterate(1L, i -> i + 1)
            .limit(N)
            .parallel()
            .reduce(0L, Long::sum);
}
```

`parallelStreamSum` 方法遇到的可能是最糟糕的情况，因为它返回 `Stream<Long>` 而非 `LongStream`，且需要处理由 `iterate` 方法产生的数据集合，而它不易分解。

而例 9-11 使用 `LongStream` 接口（包括一个 `sum` 方法）定义的 `rangeClosed` 方法，有助于 Java 分区。

例 9-11　`LongStream` 的应用

```
public long sequentialLongStreamSum() {
    return LongStream.rangeClosed(1, N)
```

注 10：作者为 Urma、Fusco 与 Mycroft，由 Manning Publications 于 2018 年 5 月出版。（该书第 1 版中文版书名为《Java 8 实战》，已由人民邮电出版社出版，此版中文版也即将推出。——译者注）

```
            .sum();
}

public long parallelLongStreamSum() {
    return LongStream.rangeClosed(1, N)
            .parallel()
            .sum();
}
```

利用 JHM 对 1000 万个元素（`N = 10,000,000`）进行测试，几种操作的测试结果如下：

```
Benchmark                Mode  Cnt   Score   Error  Units
iterativeSum             avgt   40   6.441 ± 0.019  ms/op
sequentialStreamSum      avgt   40  90.468 ± 0.613  ms/op
parallelStreamSum        avgt   40  99.148 ± 3.065  ms/op
sequentialLongStreamSum  avgt   40   6.191 ± 0.248  ms/op
parallelLongStreamSum    avgt   40   6.571 ± 2.756  ms/op
```

可以看到，装箱和拆箱操作引入了不少开销。使用 `Stream<Long>`（而非 `LongStream`）的 `parallelStreamSum` 和 `sequentialStreamSum` 操作非常慢，加之由 `iterate` 方法产生的数据集合不易分解，速度问题更加突出。`LongStream.rangeClosed` 方法则快得多，`sequentialLongStreamSum` 和 `parallelLongStreamSum` 操作在性能上几乎没有差别。

9.3 调整线程池大小

问题

用户希望在通用线程池中自定义线程数量，而不是使用默认大小的线程池。

方案

适当调整系统参数，或将任务提交到自定义的 `ForkJoinPool` 实例。

讨论

根据 Javadoc 对 `java.util.concurrent.ForkJoinPool` 类的描述，可以通过以下三种系统属性控制通用线程池的构建：

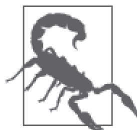
- `java.util.concurrent.ForkJoinPool.common.parallelism`
- `java.util.concurrent.ForkJoinPool.common.threadFactory`
- `java.util.concurrent.ForkJoinPool.common.exceptionHandler`

之前曾经提到过，默认情况下，通用线程池大小等于 JVM 上可用的处理器数量，它由 `Runtime.getRuntime().availableProcessors()` 确定。`parallelism` 标志用于指定并行级别（parallelism level），它是一个非负整数，可以通过编程方式或命令行方式来设置。例 9-12 显示了如何采用 `System.setProperty` 方法创建所需的并行级别。

例 9-12　采用编程方式设置通用线程池大小

```
System.setProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism", "20");
long total = LongStream.rangeClosed(1, 3_000_000)
    .parallel()
    .sum();
int poolSize = ForkJoinPool.commonPool().getPoolSize();
System.out.println("Pool size: " + poolSize); ❶
```

❶ 打印 `Pool size: 20`



将通用线程池大小设置为大于可用的 CPU 核心数无助于性能提升[11]。

在命令行中，可以像使用任何系统属性一样使用 `-D` 标志。请注意，编程设置将覆盖命令行设置，如例 9-13 所示。

例 9-13　采用系统参数设置通用线程池大小

```
$ java -cp build/classes/main concurrency.CommonPoolSize
Pool size: 20

// 注释掉System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "20");
$ java -cp build/classes/main concurrency.CommonPoolSize
Pool size: 7

$ java -cp build/classes/main \
    -Djava.util.concurrent.ForkJoinPool.common.parallelism=10 \
    concurrency.CommonPoolSize
Pool size: 10
```

本例在一台 8 核计算机上运行。默认的线程池大小为 7，但如果将 `main` 线程包括在内，则默认情况下共有 8 个活动线程。

自定义ForkJoinPool

`ForkJoinPool` 类定义了一个构造函数，它传入一个整数作为并行级别。我们可以借此创建有别于通用线程池的自定义线程池，并将任务提交给它。

例 9-14 显示了如何创建自定义线程池。

例 9-14　创建自定义 ForkJoinPool

```
ForkJoinPool pool = new ForkJoinPool(15);         ❶
ForkJoinTask<Long> task = pool.submit(            ❷
    () -> LongStream.rangeClosed(1, 3_000_000)
                .parallel()
                .sum());
try {
```

注 11：关于不同并发实现的性能比较，Alex Zhitnitsky 在 *Fork/Join Framework vs. Parallel Streams vs. ExecutorService: The Ultimate Fork/Join Benchmark* 一文中做了深入讨论。——译者注

```
        total = task.get();                        ❸
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    } finally {
        pool.shutdown();
    }
    poolSize = pool.getPoolSize();
    System.out.println("Pool size: " + poolSize);  ❹
```

❶ 实例化一个大小为 15 的 ForkJoinPool

❷ 提交 Callable<Long> 作为任务

❸ 执行任务并等待回复

❹ 打印 Pool size: 15

大部分情况下，在流上调用 parallel 方法时所用的通用线程池都能满足要求。如果需要改变其大小，请调整系统属性；如果仍然无法满足要求，请尝试创建自定义 ForkJoinPool 并将任务提交给它。

无论如何，在确定采用何种长期解决方案之前，请务必收集相关的性能数据。

另见

通过自定义线程池实现并行计算的另一种方案是采用 CompletableFuture，相关讨论请参见范例 9.5。

9.4　Future接口

问题

用户希望执行表示异步计算结果、检查计算是否完成、在必要时取消计算、检索计算结果等操作。

方案

使用能实现 java.util.concurrent.Future 接口的 CompletableFuture 类。

讨论

在本书介绍的 Java 8 和 Java 9 新特性中，CompletableFuture 是一种非常有用的类。由于 CompletableFuture 类可以实现 Future 接口，我们有必要对这种接口的用法做一简要回顾。

Java 5 引入的 java.util.concurrent 包让开发人员可以在更高的抽象层次上操作并发，而不仅限于使用简单的 wait 和 notify 原语。java.util.concurrent 包定义了一种名为 ExecutorService 的接口，其 submit 方法传入 Callable，并返回包装所需对象的 Future。

如例 9-15 所示，程序将任务提交给 ExecutorService 并打印字符串，然后在 Future 中检索值。

例 9-15　提交 Callable 并返回 Future

```
ExecutorService service = Executors.newCachedThreadPool();
Future<String> future = service.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        Thread.sleep(100);
        return "Hello, World!";
    }
});
System.out.println("Processing...");
getIfNotCancelled(future);
```

例 9-16 显示了 `getIfNotCancelled` 方法的应用。

例 9-16　在 Future 中检索值

```
public void getIfNotCancelled(Future<String> future) {
    try {
        if (!future.isCancelled()) {            ❶
            System.out.println(future.get());   ❷
        } else {
            System.out.println("Cancelled");
        }
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

❶ 检查 `Future` 的状态

❷ 通过阻塞调用以检索 `Future` 的值

在本例中，`isCancelled` 方法的用途不言自明；`get` 方法用于在 `Future` 中检索值，该方法属于阻塞调用（blocking call），返回的是其内部的泛型类型；`getIfNotCancelled` 方法采用 try/catch 代码块处理声明的异常。

输出结果如下：

```
Processing...
Hello, World!
```

由于所提交的调用会立即返回 `Future<String>`，程序将马上打印“Processing...”。之后调用 `get` 代码块直到 `Future` 完成，并打印结果。

既然本书重点讨论 Java 8，我们不妨采用 lambda 表达式替换 `Callable` 接口的匿名内部类实现，如例 9-17 所示。

例 9-17　使用 lambda 表达式并检查 Future 是否完成

```
future = service.submit(() -> {        ❶
    Thread.sleep(10);
    return "Hello, World!";
});

System.out.println("More processing...");
```

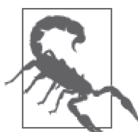
```
while (!future.isDone()) {                ❷
    System.out.println("Waiting...");
}

getIfNotCancelled(future);
```

❶ 使用 lambda 表达式替换 `Callable`

❷ 等待 `Future` 完成

可以看到，除使用 lambda 表达式之外，程序还在 `while` 循环中调用 `isDone` 方法以轮询 `Future`，直至它完成。



在循环中使用 `isDone` 方法称为**忙等待**（busy waiting），由于该方法可能产生数百万次调用，通常属于应该避免的操作。[12] `CompletableFuture` 类（详见范例 9.5、范例 9.6 与范例 9.7）可以在 `Future` 完成时提供更好的处理方式。

例 9-17 的输出结果如下：

```
More processing...
Waiting...
Waiting...
Waiting...
// 一直等待
Waiting...
Waiting...
Hello, World!
```

当某个 `Future` 完成时，程序显然需要一种更有效的方式来通知开发人员，计划将这个 `Future` 的结果用于其他操作时更是如此。`CompletableFuture` 类的作用就在于此。

此外，可以通过 `Future` 接口定义的 `cancel` 方法取消操作，如例 9-18 所示。

例 9-18　取消 `Future`

```
future = service.submit(() -> {
    Thread.sleep(10);
    return "Hello, World!";
});

future.cancel(true);

System.out.println("Even more processing...");

getIfNotCancelled(future);
```

输出结果如下：

```
Even more processing...
Cancelled
```

注 12：某些情况下可能需要忙等待，如广泛用于操作系统内核的自旋锁（spinlock）。不过，自旋锁的持有时间过长会阻止其他线程运行，导致系统性能降低。——译者注

由于 `CompletableFuture` 类实现了 `Future` 接口，本范例讨论的所有方法同样适用于 `CompletableFuture` 类。

另见

有关 `CompletableFuture` 的讨论请参见范例 9.5、范例 9.6 与范例 9.7。

9.5 完成CompletableFuture

问题

用户希望显式地完成 `CompletableFuture`，要么设置它的值，要么在调用 `get` 方法时使其抛出异常。

方案

使用 `CompletableFuture` 类定义的 `completedFuture`、`complete` 或 `completeExceptionally` 方法。

讨论

`CompletableFuture` 类不仅能实现 `Future` 接口，也能实现 `CompletionStage` 接口，它定义的数十种方法可以满足各种需要。

`CompletableFuture` 类的最大优势在于无须编写嵌套回调（nested callback）就能协调操作，相关讨论请参见范例 9.6 和范例 9.7。本范例将探讨如何在已有返回值时完成 `CompletableFuture`。

假设我们的应用程序需要根据产品 ID 检索相关产品。由于涉及某种形式的远程访问，检索过程可能会引入大量开销。这些开销包括对 RESTful Web 服务的网络调用、数据库调用以及其他相对耗时的操作。

为此，我们采用映射的形式在本地创建产品缓存：请求某个产品时，系统首先在映射中检索。如果返回 `null`，再进行开销较大的操作。例 9-19 显示了通过本地和远程两种方式来检索产品。

例 9-19　产品检索

```
private Map<Integer, Product> cache = new HashMap<>();
private Logger logger = Logger.getLogger(this.getClass().getName());

private Product getLocal(int id) {
    return cache.get(id);                                ❶
}

private Product getRemote(int id) {
    try {
        Thread.sleep(100);                               ❷
        if (id == 666) {
            throw new RuntimeException("Evil request");  ❸
```

```
        }
    } catch (InterruptedException ignored) {
    }
    return new Product(id, "name");
}
```

❶ 立即返回，但可能为 `null`

❷ 人为引入延迟，然后检索

❸ 模拟网络、数据库或其他形式的错误

接下来，我们创建一个名为 `getProduct` 的方法，传入产品 ID 作为参数，并返回相应的产品。不过，如果将返回类型设置为 `CompletableFuture<Product>`，`getProduct` 方法将立即返回；而在实际的检索过程中，程序还可以处理其他任务。

为此，需要通过某种方式来完成 `CompletableFuture`。`CompletableFuture` 类定义了三种相关的方法：

```
           boolean              complete(T value)
static <U> CompletableFuture<U> completedFuture(U value)
           boolean              completeExceptionally(Throwable ex)
```

如果已有 `CompletableFuture` 且希望将其设置为给定的值，可以采用 `complete` 方法；`CompletableFuture` 方法是一种工厂方法，它使用给定的值创建一个新的 `CompletableFuture`；`completeExceptionally` 方法使用给定的异常来结束 `Future`。

例 9-20 显示了上述三种方法的应用。程序假定存在一种从远程系统返回产品的遗留机制，希望使用这种机制完成 `Future`。

例 9-20　完成 `CompletableFuture`

```
public CompletableFuture<Product> getProduct(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            return CompletableFuture.completedFuture(product);   ❶
        } else {
            CompletableFuture<Product> future = new CompletableFuture<>();
            Product p = getRemote(id);                            ❷
            cache.put(id, p);
            future.complete(p);                                   ❸
            return future;
        }
    } catch (Exception e) {
        CompletableFuture<Product> future = new CompletableFuture<>();
        future.completeExceptionally(e);                          ❹
        return future;
    }
}
```

❶ 在缓存中检索到产品（如果有的话）后完成

❷ 遗留检索方式

❸ 在执行遗留检索方式后完成（异步操作的情况参见例 9-22）

❹ 如果出现问题，在抛出异常后完成

在本例中，`getProduct` 方法首先尝试在缓存中检索产品。如果映射返回非空值，则使用工厂方法 `completedFuture` 返回该值。

如果缓存返回 `null`，说明需要进行远程访问。程序模拟了一种可能是遗留代码的同步方案，稍后将详细讨论。`CompletableFuture` 类被实例化，`complete` 方法采用生成的值进行填充。

最后，如果出现严重问题（将 ID 设置为 666 以便模拟），程序将抛出 `RuntimeException`。`completeExceptionally` 方法传入 `RuntimeException` 作为参数，并使用该异常结束 `Future`。

例 9-21 的测试用例显示了异常处理的工作方式。

例 9-21　`completeExceptionally` 方法的应用

```
@Test(expected = ExecutionException.class)
public void testException() throws Exception {
    demo.getProduct(666).get();
}

@Test
public void testExceptionWithCause() throws Exception {
    try {
        demo.getProduct(666).get();
        fail("Houston, we have a problem...");
    } catch (ExecutionException e) {
        assertEquals(ExecutionException.class, e.getClass());
        assertEquals(RuntimeException.class, e.getCause().getClass());
    }
}
```

上述两项测试均能通过。在 `CompletableFuture` 上调用 `completeExceptionally` 方法时，`get` 方法将抛出 `ExecutionException`，这是由首先导致问题的异常（本例为 `RuntimeException`）所引起的。



`get` 方法抛出的 `ExecutionException` 是一个受检异常（checked exception）。`join` 方法与 `get` 方法相同，不同之处在于，如果异常完成，`join` 方法将抛出由底层异常所引发的 `CompletionException`，它是一个非受检异常（unchecked exception）。

在例 9-20 中，最有可能替换的是执行产品同步检索的那部分代码。为此，可以使用 `CompletableFuture` 类定义的另一种静态工厂方法 `supplyAsync`，它包括以下形式：

```
static     CompletableFuture<Void> runAsync(Runnable runnable)
static     CompletableFuture<Void> runAsync(Runnable runnable,
                                            Executor executor)

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,
                                            Executor executor)
```

supplyAsync 方法返回一个使用给定 Supplier 的对象；如果不需要返回对象，则使用 runAsync 方法更方便。两种方法的单参数形式使用默认的通用 fork/join 线程池，而双参数形式使用给定的执行器（executor）作为第二个参数。

例 9-22 显示了采用异步模式检索产品。

例 9-22　利用 supplyAsync 方法检索产品

```
public CompletableFuture<Product> getProductAsync(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            logger.info("getLocal with id=" + id);
            return CompletableFuture.completedFuture(product);
        } else {
            logger.info("getRemote with id=" + id);

            return CompletableFuture.supplyAsync(() -> {    ❶
                Product p = getRemote(id);
                cache.put(id, p);
                return p;
            });
        }
    } catch (Exception e) {
        logger.info("exception thrown");
        CompletableFuture<Product> future = new CompletableFuture<>();
        future.completeExceptionally(e);
        return future;
    }
}
```

❶ 与之前的操作相同，但采用异步模式返回检索到的产品

可以看到，程序在实现 Supplier<Product> 的 lambda 表达式中检索产品。我们总是可以将其作为独立的方法提取出来，并将代码简化为一个方法引用。

不过，如何在 CompletableFuture 完成之后调用其他操作颇费周章。有关多个 CompletableFuture 实例之间的协调请参见范例 9.6。

另见

本范例的示例以 Kenneth Jørgensen 在其博文中讨论的一个类似示例为基础。[13]

9.6　多个CompletableFuture之间的协调（第1部分）

问题

用户希望在一个 Future 完成之后触发另一个动作（action）。

注 13：参见 Kenneth Jørgensen 博客上题为“Introduction to CompletableFutures”的博文。

方案

使用 CompletableFuture 类中用于协调动作的各种实例方法，如 thenApply、thenCompose、thenRun 等。

讨论

CompletableFuture 类的最大优势在于能很容易地将多个 Future 链接起来。我们可以创建多个 Future 以表示需要执行的各种任务，然后通过一个 Future 的完成来触发另一个 Future 的执行，达到协调动作的目的。

我们考虑如何实现下面这个简单的任务：

- 向 Supplier 请求一个包含数字的字符串
- 将数字解析为整数
- 将整数倍增
- 打印倍增后的结果

相关实现如例 9-23 所示，程序并不复杂。

例 9-23　利用 CompletableFuture 协调多个任务

```
private String sleepThenReturnString() {
    try {
        Thread.sleep(100);                   ❶
    } catch (InterruptedException ignored) {
    }
    return "42";
}

CompletableFuture.supplyAsync(this::sleepThenReturnString)
        .thenApply(Integer::parseInt)        ❷
        .thenApply(x -> 2 * x)               ❷
        .thenAccept(System.out::println)     ❸
        .join();                             ❹
System.out.println("Running...");
```

❶ 人为引入延迟

❷ 在前一阶段完成后应用 Function

❸ 在前一阶段完成后应用 Consumer

❹ 检索完成的结果

由于对 join 的调用属于阻塞调用，执行上述程序将输出 84 并后跟“Running...”。supplyAsync 方法传入 Supplier（本例为 String 类型）。thenApply 方法传入 Function，其输入参数为前一个 CompletionStage 的结果。其中第一个 thenApply 方法中的函数将字符串转换为一个整数，第二个 thenApply 方法中的函数将整数倍增。最后，thenAccept 方法传入 Consumer，在前一个阶段完成后执行。

CompletableFuture 类定义了多种不同的协调方法，完整列表（不包括重载形式，将在之后讨论）如表 9-1 所示。

表9-1：CompletableFuture类定义的协调方法

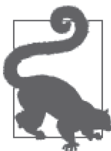
修饰符	返回类型	方法名	参数
	CompletableFuture<Void>	acceptEither	CompletionStage<? extends T> other, Consumer<? super T> action
static	CompletableFuture<Void>	allOf	CompletableFuture<?>... cfs
static	CompletableFuture<Object>	anyOf	CompletableFuture<?>... cfs
<U>	CompletableFuture<U>	applyToEither	CompletionStage<? extends T> other, Function<? super T, U> fn
	CompletableFuture<Void>	runAfterBoth	CompletionStage<?> other, Runnable action
	CompletableFuture<Void>	thenAccept	Consumer<? super T> action
<U>	CompletableFuture<U>	thenApply	Function<? super T> action, ? extends U> fn
<U,V>	CompletableFuture<V>	thenCombine	CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn
<U>	CompletableFuture<U>	thenCompose	Function<? super T, ? extends CompletionStage<U>> fn
	CompletableFuture<Void>	thenRun	Runnable action
	CompletableFuture<T>	whenComplete	BiConsumer<? super T, ? super Throwable> action

上表列出的所有方法均使用工作者线程（worker thread）的通用 ForkJoinPool，其大小与处理器数量相等。前面已经讨论过工厂方法 runAsync 和 supplyAsync，二者指定 Runnable 或 Supplier，并返回 CompletableFuture。如上表所示，可以通过链接其他方法（如 thenApply 或 thenCompose）来添加将在前一个任务完成后开始执行的任务。

表 9-1 并未列出每种方法包含的其他两种模式，二者以 Async 结尾，一种传入 Executor，而另一种不传入。以 thenAccept 方法为例，其完整形式如下：

```
CompletableFuture<Void> thenAccept(Consumer<? super T> action)
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)
CompletableFuture<Void> thenAcceptAsync(
  Consumer<? super T> action, Executor executor)
```

第一种形式在与原有任务相同的线程中执行其 Consumer 参数；第二种形式将 Consumer 再次提交给线程池；第三种形式提供一个 Executor，用于运行任务而非通用 fork/join 线程池。



是否采用这些方法的 Async 形式应视情况而定。尽管异步操作能加快单个任务的执行速度，但引入的开销可能无助于整体速度的提升。

由于 ExecutorService 接口实现了 Executor 接口，我们也可以使用自定义 Executor 而不是通用线程池。例 9-24 显示了利用单独的线程池执行 CompletableFuture 任务。

例 9-24　在单独的线程池中运行 CompletableFuture 任务

```
ExecutorService service = Executors.newFixedThreadPool(4);
CompletableFuture.supplyAsync(this::sleepThenReturnString, service)     ❶
        .thenApply(Integer::parseInt)
        .thenApply(x -> 2 * x)
        .thenAccept(System.out::println)
        .join();
System.out.println("Running...");
```

❶ 提供单独的线程池作为参数

在本例中，thenApply 和 thenAccept 方法使用的线程与 supplyAsync 方法相同。不过，如果使用 thenApplyAsync 方法，那么除非添加另一个线程池作为附加参数，否则任务将被提交到线程池。

在通用 ForkJoinPool 中等待完成

默认情况下，CompletableFuture 使用所谓的“通用”fork/join 线程池，后者是一种经过优化并执行**工作窃取**（work stealing）的线程池。根据 Javadoc 的描述，这种线程池中的所有线程“尝试查找并执行提交到线程池或由其他活动任务创建的任务”。需要注意的是，所有工作者线程均为**守护**线程（daemon thread）：如果在线程结束前退出程序，线程将终止。

换言之，在执行例 9-23 所示的代码时，如果没有调用 join 方法，程序将只打印“Running...”，而不会输出 Future 的结果，系统在任务完成前即告终止。

可以采用两种方案解决这个问题。一种方案是调用 get 或 join 方法，二者将阻塞调用进程，直至检索到结果。另一种方案是为通用线程池设置一个超时时间（time-out period），告诉程序等待直至所有线程执行完毕：

```
ForkJoinPool.commonPool().awaitQuiescence(long timeout, TimeUnit unit)
```

如果将线程池的等待周期设置得足够长，Future 就会完成。awaitQuiescence 方法用于通知系统等待，直至所有工作者线程空闲，或所设置的超时时间结束（以先到者为准）。

对于返回值的 CompletableFuture 实例，可以通过 get 或 join 方法检索该值。两种方法属于阻塞调用，直至 Future 完成或抛出异常。不同之处在于，get 方法抛出的是受检异常 ExecutionException，而 join 方法抛出的是非受检异常 CompletionException，因此在 lambda 表达式中更容易使用 join 方法。

cancel 方法用于取消 CompletableFuture，该方法传入一个布尔值作为参数：

```
boolean cancel(boolean mayInterruptIfRunning)
```

如果 CompletableFuture 尚未完成，cancel 方法将利用 CancellationException 使其完成，所有依赖的 CompletableFuture 也会由于 CancellationException 引发的 CompletionException

而异常完成。此时，布尔参数不执行任何操作。[14]

之前的示例介绍了 thenApply 和 thenAccept 方法的应用。而 thenCompose 是一种实例方法，可以将某个 Future 链接到原有 Future，使得第一个 Future 的结果也能用于第二个 Future。thenCompose 方法的应用如例 9-25 所示，这可能是实现两个数相加的最复杂的程序了。

例 9-25　两个 Future 的复合

```
@Test
public void compose() throws Exception {
    int x = 2;
    int y = 3;
    CompletableFuture<Integer> completableFuture =
        CompletableFuture.supplyAsync(() -> x)
            .thenCompose(n -> CompletableFuture.supplyAsync(() -> n + y));

    assertTrue(5 == completableFuture.get());
}
```

thenCompose 方法的参数是一个函数，它传入第一个 Future 的结果，并将其转换为第二个 Future 的输出。不过，如果希望两个 Future 彼此独立，可以改用 thenCombine 方法，如例 9-26 所示。[15]

例 9-26　两个 Future 的合并

```
@Test
public void combine() throws Exception {
    int x = 2;
    int y = 3;
    CompletableFuture<Integer> completableFuture =
        CompletableFuture.supplyAsync(() -> x)
            .thenCombine(CompletableFuture.supplyAsync(() -> y),
                         (n1, n2) -> n1 + n2);

    assertTrue(5 == completableFuture.get());
}
```

thenCombine 方法传入 Future 和 BiFunction 作为参数。计算结果时，两个 Future 的结果都能在函数中使用。

CompletableFuture 类还定义了一种名为 handle 的方法，其签名如下：

```
<U> CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn)
```

BiFunction 的两个输入参数为 Future 的结果（正常完成）和抛出的异常（异常结束），返回的参数由程序决定。handle 方法也有两种 Async 模式，一种传入 BiFunction，另一种传入 BiFunction 和 Executor。handle 方法的应用如例 9-27 所示。

例 9-27　handle 方法的应用

```
private CompletableFuture<Integer> getIntegerCompletableFuture(String num) {
    return CompletableFuture.supplyAsync(() -> Integer.parseInt(num))
```

注 14：有趣的是，根据 Javadoc 的描述，参数 mayInterruptIfRunning“不起作用，因为中断不用于控制处理”。

注 15：好吧，这同样可能是实现两个数字相加的最复杂的程序。

```
            .handle((val, exc) -> val != null ? val : 0);
    }

    @Test
    public void handleWithException() throws Exception {
        String num = "abc";
        CompletableFuture<Integer> value = getIntegerCompletableFuture(num);
        assertTrue(value.get() == 0);
    }

    @Test
    public void handleWithoutException() throws Exception {
        String num = "42";
        CompletableFuture<Integer> value = getIntegerCompletableFuture(num);
        assertTrue(value.get() == 42);
    }
```

本例解析字符串以查找相应的整数。解析成功则返回整数，解析失败则抛出 `ParseException`，`handle` 方法返回 0。上述两个测试表明，`handle` 方法能正确处理这两种情况。

从上面这些示例不难看到，可以通过多种方式在通用线程池或自定义执行器中同步或异步地合并多个任务。范例 9.7 将给出一个更复杂的示例，讨论如何协调多个 `CompletableFuture`。

另见

有关如何协调多个 `CompletableFuture`，范例 9.7 给出了一个更复杂的示例。

9.7　多个CompletableFuture之间的协调（第2部分）

问题

用户希望通过更复杂的示例了解如何协调多个 `CompletableFuture` 实例。

方案

在美国职棒大联盟（MLB）赛季的每个比赛日访问官方网站，其中包括指向当日比赛的链接。下载每场比赛的技术统计信息（box score），并将其转换为一个 Java 类。采用异步方式保存数据后计算每场比赛的结果，找出总分最高的比赛，然后打印最高分以及出现最高分的那场比赛。

讨论

较之其他简单的示例，本范例所讨论的应用程序更为复杂。希望读者能从中受到启发，理解如何通过合并多个 `CompletableFuture` 任务来完成工作。

MLB 官方网站保存了指定比赛日中每场比赛的得分，我们的应用程序即以此为基础。[16] 以 2017 年 6 月 14 日为例，包括当日所有比赛信息的页面如图 9-1 所示。

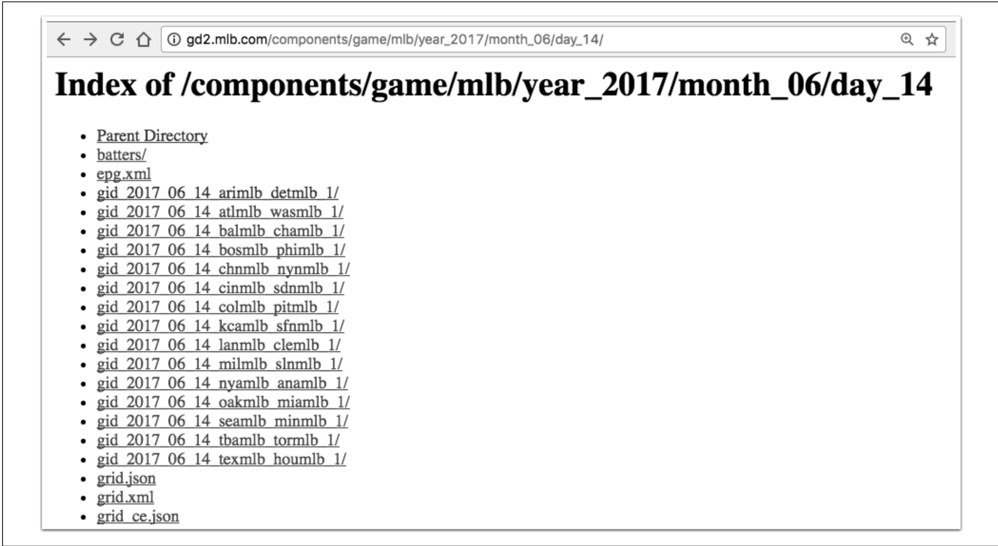


图 9-1：2017 年 6 月 14 日进行的比赛

在上述页面中，每个链接指向一场比赛。链接以字母 gid 开头，后跟年、月、日以及主队和客队代码。点击某个链接，跳转后的页面包含一个文件列表，其中有一个名为 boxscore.json 的文件。

我们的应用程序将完成以下任务。

(1) 访问提供指定日期范围内各场比赛信息的网站；
(2) 确定每个页面的比赛链接；
(3) 下载每场比赛的 boxscore.json 文件；
(4) 将 boxscore.json 文件转换为相应的 Java 对象；
(5) 将下载结果保存到本地文件；
(6) 计算每场比赛的得分；
(7) 检索总分最高的比赛；
(8) 打印所有比赛的得分，以及出现最高得分的比赛及其得分。

可以将大部分任务安排为并发执行，不少任务都能以并行方式运行。

受篇幅所限，无法将完整的程序代码复制到书中，不过读者可以从 GitHub 下载 [17]。本范例将重点讨论并行流和 `CompletableFuture` 的应用。

注 16：只要了解以下内容，理解本例就不会存在障碍：在棒球比赛中，两队轮流攻守，得分较高的队胜出，比赛的统计数据称为技术统计。

注 17：链接如下：https://github.com/kousen/cfboxscores。

第一个难点在于如何找出给定范围内每个比赛日的比赛链接。如例 9-28 所示，GamePageLinksSupplier 类实现了 Supplier 接口，其作用是生成一个表示比赛链接的字符串列表。

例 9-28　获取某个日期范围内的比赛链接

```
public class GamePageLinksSupplier implements Supplier<List<String>> {
    private static final String BASE =
        "http://gd2.mlb.com/components/game/mlb/";
    private LocalDate startDate;
    private int days;

    public GamePageLinksSupplier(LocalDate startDate, int days) {
        this.startDate = startDate;
        this.days = days;
    }

    public List<String> getGamePageLinks(LocalDate localDate) {
        // 使用Jsoup库解析HTML网页，并提取以"gid"开头的链接
    }

    @Override
    public List<String> get() {        ❶
        return Stream.iterate(startDate, d -> d.plusDays(1))
            .limit(days)
            .map(this::getGamePageLinks)
            .flatMap(list -> list.isEmpty() ? Stream.empty() : list.stream())
            .collect(Collectors.toList());
    }
}
```

❶ Supplier<List<String>> 所需的方法

get 方法使用 Stream.iterate 方法对某个范围内的日期进行迭代：从给定日期开始，逐天递增直至上限。



Java 9 为 LocalDate 类引入了 datesUntil 方法，它将生成 Stream<LocalDate>。相关讨论请参见范例 10.7。

每个 LocalDate 都成为 getGamePageLinks 方法的参数，它使用 Jsoup 库解析 HTML 页面，并查找所有以 gid 开头的链接，然后以字符串的形式返回这些链接。

接下来，程序通过实现 Function 接口的 BoxscoreRetriever 类来访问每个比赛链接中的 boxscore.json 文件，如例 9-29 所示。

例 9-29　在比赛链接列表中检索技术统计列表

```
public class BoxscoreRetriever implements Function<List<String>, List<Result>> {
    private static final String BASE =
        "http://gd2.mlb.com/components/game/mlb/";

    private OkHttpClient client = new OkHttpClient();
    private Gson gson = new Gson();
```

```
    @SuppressWarnings("ConstantConditions")
    public Optional<Result> gamePattern2Result(String pattern) {
        // 省略代码
        String boxscoreUrl = BASE + dateUrl + pattern + "boxscore.json";

        // 设置OkHttp库以创建网络调用
        try {
            // 获取响应
            if (!response.isSuccessful()) {
                System.out.println("Box score not found for " + boxscoreUrl);
                return Optional.empty();      ❶
            }

            return Optional.ofNullable(
                gson.fromJson(response.body().charStream(), Result.class)); ❷
        } catch (IOException e) {
            e.printStackTrace();
            return Optional.empty();          ❶
        }
    }

    @Override
    public List<Result> apply(List<String> strings) {
        return strings.parallelStream()
                .map(this::gamePattern2Result)
                .filter(Optional::isPresent)
                .map(Optional::get)
                .collect(Collectors.toList());
    }
}
```

❶ 如果由于降雨或其他因素而未能找到技术统计信息，则返回空 `Optional`

❷ 利用 Gson 库将 JSON 转换为 `Result`

`BoxscoreRetriever` 类需要使用 OkHttp 库和 Gson 库以下载 JSON 格式的技术统计信息，并将其转换为 `Result` 类型的对象。由于 `BoxscoreRetriever` 类实现了 `Function` 接口，可以实现 `apply` 方法，从而将字符串列表转换为结果列表。如果给定的比赛由于降雨而取消，或因为某些原因导致网络连接中断，则可能找不到该场比赛的技术统计。这种情况下，`gamePattern2Result` 方法将返回一个为空的 `Optional<Result>`。

`apply` 方法读取各场比赛的链接，将它们转换为相应的 `Optional<Result>`。接下来，`apply` 方法对流进行筛选，仅传递非空的 `Optional` 实例，然后在这些 `Optional` 实例上调用 `get` 方法，最后将它们收集到结果列表。



Java 9 为 `Optional` 类引入了 `stream` 方法，它可以将 `filter(Optional::isPresent)` 和 `map(Optional::get)` 简化为 `flatMap(Optional::stream)`。相关讨论请参见范例 10.6。

检索到技术统计信息后可以将其保存为本地文件，如例 9-30 所示。

例 9-30　将每场比赛的技术统计信息保存为文件

```
private void saveResultList(List<Result> results) {
    results.parallelStream().forEach(this::saveResultToFile);
}

public void saveResultToFile(Result result) {
    // 根据比赛日期和队名确定文件名
    try {
        File file = new File(dir + "/" + fileName);
        Files.write(file.toPath().toAbsolutePath(),      ❶
                    gson.toJson(result).getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

❶ 创建或覆盖文件，然后将其关闭

如果文件不存在，`Files.write` 方法（使用默认参数）将创建一个新文件，否则覆盖原有文件。创建或覆盖文件后将其关闭。

程序还使用其他两种后期处理方法：`getMaxScore` 用于确定某场给定比赛的最高总分，而 `getMaxGame` 将返回出现最高分的那场比赛。两种方法的应用如例 9-31 所示。

例 9-31　获取最高总分以及出现最高分的比赛

```
private int getTotalScore(Result result) {
    // 两队得分之和
}

public OptionalInt getMaxScore(List<Result> results) {
    return results.stream()
            .mapToInt(this::getTotalScore)
            .max();
}

public Optional<Result> getMaxGame(List<Result> results) {
    return results.stream()
            .max(Comparator.comparingInt(this::getTotalScore));
}
```

最后，通过 `CompletableFuture` 将前面讨论的所有方法与类合并在一起。主程序代码如例 9-32 所示。

例 9-32　主程序代码

```
public void printGames(LocalDate startDate, int days) {
    CompletableFuture<List<Result>> future =
        CompletableFuture.supplyAsync(
            new GamePageLinksSupplier(startDate, days))
                .thenApply(new BoxscoreRetriever());        ❶

    CompletableFuture<Void> futureWrite =
        future.thenAcceptAsync(this::saveResultList)        ❷
            .exceptionally(ex -> {
```

```
                System.err.println(ex.getMessage());
                return null;
            });

        CompletableFuture<OptionalInt> futureMaxScore =
            future.thenApplyAsync(this::getMaxScore);
        CompletableFuture<Optional<Result>> futureMaxGame =
            future.thenApplyAsync(this::getMaxGame);
        CompletableFuture<String> futureMax =
            futureMaxScore.thenCombineAsync(futureMaxGame,      ❸
                (score, result) ->
                    String.format("Highest score: %d, Max Game: %s",
                                  score.orElse(0), result.orElse(null)));

        CompletableFuture.allOf(futureWrite, futureMax).join(); ❹

        future.join().forEach(System.out::println);
        System.out.println(futureMax.join());
    }
```

❶ 检索技术统计信息的协调任务

❷ 保存为文件，如果出现问题则异常完成

❸ 合并最高总分与出现最高分的比赛这两个任务

❹ 完成所有任务

可以看到，程序创建了多个 `CompletableFuture` 实例。第一个 `CompletableFuture` 实例使用 `GamePageLinksSupplier` 类检索指定日期内所有比赛的页面链接，然后通过 `BoxscoreRetriever` 类将这些链接转换为结果。第二个 `CompletableFuture` 实例设置将结果写入磁盘，如果出现问题则异常完成。两个后期处理方法 `getMaxScore` 和 `getMaxGame` 分别用于查找最高总分以及出现最高分的那场比赛，[18] 而 `allOf` 方法用于完成所有任务。最后，程序打印相应的结果。

注意 `thenApplyAsync` 方法的应用。该方法并非必需，但能使任务以异步方式运行。

如果需要检索 2017 年 5 月 5 日到 5 月 7 日三天的技术统计信息，请使用以下语句：

```
GamePageParser parser = new GamePageParser();
parser.printGames(LocalDate.of(2017, Month.MAY, 5), 3);
```

输出结果如下：

```
Box score not found for Los Angeles at San Diego on May 5, 2017
May 5, 2017: Arizona Diamondbacks 6, Colorado Rockies 3
May 5, 2017: Boston Red Sox 3, Minnesota Twins 4
May 5, 2017: Chicago White Sox 2, Baltimore Orioles 4
// 更多数据
May 7, 2017: Toronto Blue Jays 2, Tampa Bay Rays 1
May 7, 2017: Washington Nationals 5, Philadelphia Phillies 6
Highest score: 23, Max Game: May 7, 2017: Boston Red Sox 17, Minnesota Twins 6
```

注 18：这两项操作显然可以一起完成，程序将二者分开是为了展示 `thenCombine` 的应用。

希望本范例能对读者有所启发，通过综合运用本书介绍的各种知识，包括 `Future` 任务（使用 `CompletableFuture`）、函数式接口（如 `Supplier` 和 `Function`）、类（如 `Optional`、`Stream` 与 `LocalDate`）以及方法（如 `map`、`filter` 与 `flatMap`），掌握如何解决复杂而有趣的问题。

另见

有关多个 `CompletableFuture` 之间的协调请参见范例 9.6。

第 10 章

Java 9新特性

写作本书时（2017 年 6 月），Java SDK 9 已达到功能完成（Feature Complete）状态，但尚未发布。[1] 在众多新特性中，外界最为关注的当属 Jigsaw 项目（Project Jigsaw），它是 Java 9 引入的一种模块化（modularization）机制。

这一章的范例将讨论 Java 9 的各种新特性，包括接口中的私有方法以及创建不可变集合所用的工厂方法，并介绍 `Stream`、`Collectors` 与 `Optional` 新增的各种方法。所有范例均在 Java SE 9 Early Access build 174 下测试通过。

另一方面，本章**不**讨论以下新特性：

- JShell 交互式控制台
- 改进的 `try-with-resources` 代码块
- 钻石运算符（diamond operator）的轻松语法
- 新增的弃用警告
- 用于实现反应式流（Reactive Streams）的 Flow API
- 用于实现栈遍历的 Stack-Walking API
- 改进的 Process API

之所以不讨论这些特性，是因为它们要么用得不多（如钻石运算符、改进的 `try-with-resources` 代码块与弃用警告），要么较为专业（如 Stack-Walking API 和改进的 Process API）。而文档和教程对 JShell 的描述已很详细，本书不再赘述。

此外，虽然反应式流的引入令人眼前一亮，但开源社区已有 Reactive Streams、RxJava 等类似的 API 存在。对于社区将以何种方式支持新的 Java 9 API，静观其变或许更为稳妥。

注 1：Java SE 9 已于 2017 年 9 月 21 日正式发布。截至 2018 年 3 月 1 日，最新版本为 9.0.4。——译者注

这一章的范例有望涵盖最常见的用例，若非如此，本书下一版将增加更多用例。[2]

此外，这一章的范例与其他章节略有不同。本书以用例驱动的形式编写和组织内容，每个范例致力于解决一个特定类型的问题。而这一章的部分范例只是对 Java 9 引入的新特性进行概述，不涉及具体类型的问题。

10.1 Jigsaw中的模块

问题

用户希望访问 Java 标准库中的模块，并将自己的代码封装为模块。

方案

学习 Jigsaw 模块的基础知识，以及如何使用经过模块化处理的 JDK。然后等待 Java 9 正式发布，再决定是否进行升级。

讨论

JSR 376 即 Java 平台模块系统（Java Platform Module System，JPMS），它堪称 Java 9 最大也是最富有争议的变革。对 Java 进行模块化处理的工作已开展近 10 年[3]，并取得了不同程度的成功，JPMS 就是这些成果的集中体现。

模块系统致力于提供“强有力”的封装，虽然这对维护有利，但其副作用也不容小觑。对一门有 20 多年历史且注重保持向后兼容性的语言来说，这种根本性的调整绝非易事。

例如，**模块**的概念改变了 `public` 和 `private` 的性质。如果模块没有导出特定的包，就无法访问包中的类（即便被声明为 `public`）。与之类似，如果某个类不在导出的包中，就不能通过反射来访问类中的非公共成员。这将影响到基于反射的库和框架（包括流行的 Spring 和 Hibernate），以及 JVM 上几乎所有的非 Java 语言。作为对各方的让步，Java 开发团队建议在 Java 9 中默认允许使用命令行标志 `--illegal-access=permit`，但在今后的版本中予以禁用。[4]

写作本书时（2017 年 6 月），将 JPMS 规范纳入 Java 9 的提案已被否决过一次，但 JSR 376 专家组正在对规范进行修改，以便为再次投票做准备。[5] 此外，Java 9 的发布日期被推迟到 2017 年 9 月下旬。[6]

注 2：那时候，Jigsaw 的细节应该已经很完善了——但愿如此。

注 3：Jigsaw 项目于 2008 年启动。

注 4：相关信息参见“Proposal: Allow illegal reflective access by default in JDK 9”。

注 5：在 2017 年 6 月 13 日到 6 月 26 日进行的第二次投票中，JPMS 规范获得了一致通过（一票弃权）。

注 6：Java 9 原定于 2016 年 9 月 22 日发布，但经历了两次重大推迟（2017 年 3 月、2017 年 7 月），主要原因在于各方对模块化的争议较大。——译者注

尽管如此，Java 9 可能会包括 Jigsaw 的部分内容，其基本功能也已确定。本范例旨在介绍这些功能，方便读者了解必要的背景信息，以便在 JPMS 规范获批后做好应用准备。

首先需要指出的是，读者不必急着将自己的代码模块化。虽然 Java 库已被模块化，其他依赖库也正在进行模块化处理，但读者不妨等到系统稳定后再对代码进行操作。

模块

除了所谓的**未命名**模块（unnamed module），Java 9 定义的模块都有一个名称，并通过名为 module-info.java 的文件定义相关的依赖和需要导出的包。此外，在模块可交付的 JAR 文件中包括一个经过编译的 module-info.class 文件。

module-info.java 文件称为**模块描述符**（module descriptor），它以关键字 `module` 开头，通过关键字 `requires` 和 `exports` 描述模块的功能。接下来，我们以一个简单的“Hello, World!”程序为例来讨论，程序将使用两个模块以及 JVM。

两个示例模块为 `com.oreilly.suppliers` 和 `com.kousenit.clients`。前者提供表示姓名的字符串流，后者将每个姓名以及欢迎消息打印到控制台。



“反向 URL”（reversed URL）模式是目前推荐使用的模块命名约定。

对于 `Supplier` 模块，`NamesSupplier` 类的源代码如例 10-1 所示。

例 10-1　提供姓名流

```
package com.oreilly.suppliers;

// 导入

public class NamesSupplier implements Supplier<Stream<String>> {
    private Path namesPath = Paths.get("server/src/main/resources/names.txt");

    @Override
    public Stream<String> get() {
        try {
            return Files.lines(namesPath);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

请注意，`Supplier` 模块保存在 IntelliJ 模块中。不过，IntelliJ IDEA 同样使用“模块”一词，但此“模块”非彼“模块”，它表示“服务器”（server），这也是文本文件的路径中出现“server”的原因。

names.txt 文件的内容如下：[7]

```
Londo
Vir
G'Kar
Na'Toth
Delenn
Lennier
Kosh
```

对于 `Client` 模块，`Main` 类的源代码如例 10-2 所示。

例 10-2　打印姓名

```
package com.kousenit.clients;

// 导入

public class Main {
    public static void main(String[] args) throws IOException {
        NamesSupplier supplier = new NamesSupplier();

        try (Stream<String> lines = supplier.get()) {  ❶
            lines.forEach(line -> System.out.printf("Hello, %s!%n", line));
        }
    }
}
```

❶ `try-with-resources` 自动关闭流

例 10-3 显示了定义 `Supplier` 模块的 module-info.java 文件。

例 10-3　定义 Supplier 模块

```
module com.oreilly.suppliers {      ❶
    exports com.oreilly.suppliers;  ❷
}
```

❶ 模块名

❷ 使模块可供其他模块使用

例 10-4 显示了定义 `Client` 模块的 module-info.java 文件。

例 10-4　定义 Client 模块

```
module com.kousenit.clients {       ❶
    requires com.oreilly.suppliers; ❷
}
```

❶ 模块名

❷ 请求 `Supplier` 模块

注 7：是时候在本书中使用《巴比伦五号》的示例了——想必空间站也是由各种模块构成的。(《巴比伦五号》是一部在 1994 年到 1998 年期间播出的美国科幻电视连续剧，共 110 集。names.txt 文件中出现的姓名均为剧中角色。——译者注)

执行例 10-2 的程序，输出如下：

```
Hello, Vir!
Hello, G'Kar!
Hello, Na'Toth!
Hello, Delenn!
Hello, Lennier!
Hello, Kosh!
```

定义 Supplier 模块时必须使用 exports 子句，以便 NamesSupplier 类对 Client 模块可见。Client 模块定义中的 requires 子句用于通知程序，Client 模块需要使用 Supplier 模块中的类。

如果希望在 Supplier 模块中记录对服务器的访问，一种方案是利用 java.util.logging 包定义的 Logger 类在 JVM 中添加一个日志记录器，如例 10-5 所示。

例 10-5　为 Supplier 模块添加日志记录

```
public class NamesSupplier implements Supplier<Stream<String>> {
    private Path namesPath = Paths.get("server/src/main/resources/names.txt");
    private Logger logger = Logger.getLogger(this.getClass().getName()); ❶

    @Override
    public Stream<String> get() {
        logger.info("Request for names on " + Instant.now());            ❷
        try {
            return Files.lines(namesPath);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

❶ 创建日志记录器

❷ 使用时间戳记录对服务器的访问

不过，上述代码无法编译。这是因为经过模块化处理后，JVM 目前默认提供的唯一模块是 java.base，但 java.base 并不包括 java.util.logging 包。为使用 Logger 类，需要更新定义 Supplier 模块的 module-info.java 文件，如例 10-6 所示。

例 10-6　定义 Supplier 模块（更新后的 module-info.java 文件）

```
module com.oreilly.suppliers {
    requires java.logging;          ❶
    exports com.oreilly.suppliers;
}
```

❶ 除 java.base 模块外，从 JVM 请求 java.logging 模块

JVM 中的每个模块都有各自的 module-info.java 文件。以 java.logging 模块为例，定义它的 module-info.java 文件如例 10-7 所示。

例 10-7　Logging API 的 module-info.java 文件

```
module java.logging {
    exports java.util.logging;
    provides jdk.internal.logger.DefaultLoggerFinder with
        sun.util.logging.internal.LoggingProviderImpl;
}
```

上述 module-info.java 文件不仅导出 `java.logging` 模块，当客户端请求日志记录器时，还会以 `LoggingProviderImpl` 类的形式提供 SPI（服务提供者接口）`DefaultLoggerFinder` 的内部实现。



Jigsaw 还提供用于处理服务定位器和提供者的机制，详细信息请参考文档。

希望本范例能对读者有所启发，了解模块是如何定义与应用的。

在 JPMS 规范获批之前，JSR 376 专家组还将解决更多与模块有关的问题，不少问题涉及遗留代码的移植。例如，未命名模块和自动模块（automatic module）的代码不属于任何模块，而是位于"模块路径"（module path）以及由现有遗留 JAR 文件所构成的模块中。有关 JPMS 的争议，相当一部分在于如何处理这些问题。

另见

Jigsaw 的开发属于 OpenJDK 项目的一部分，感兴趣的读者可以阅读快速指引（"Project Jigsaw: Module System Quick-Start Guide"）和相关文档（"The State of the Module System"）。

10.2　接口中的私有方法

问题

用户希望将私有方法添加到接口中，这些方法可以被接口中的其他方法调用。

方案

Java 9 目前支持在接口方法中使用 `private` 关键字。

讨论

从 Java 8 开始，开发人员可以为接口编写方法实现，并将它们标记为 `default` 或 `static`。接下来，将 `private` 方法添加到接口是顺理成章的事情。

私有方法使用关键字 `private`，且必须有一个实现。与类中的私有方法相似，无法重写接口中的私有方法。更重要的是，私有方法只能从同一个源文件中调用。

例 10-8 虽然略显刻意，但仍有一定说服力。

例 10-8　接口中的私有方法

```
import java.util.function.IntPredicate;
import java.util.stream.IntStream;

public interface SumNumbers {
    default int addEvens(int... nums) {
        return add(n -> n % 2 == 0, nums);
    }

    default int addOdds(int... nums) {
        return add(n -> n % 2 != 0, nums);
    }

    private int add(IntPredicate predicate, int... nums) {    ❶
        return IntStream.of(nums)
                .filter(predicate)
                .sum();
    }
}
```

❶ 私有方法

由于接口中的默认访问是公共的，addEvens 和 addOdds 方法都属于 public，它们传入整数的可变参数列表作为参数。两种方法的 default 实现被委托给 add 方法。除整数的可变参数列表外，add 方法还传入 IntPredicate 作为参数。任何客户端都无法访问被声明为私有方法的 add 方法，即便通过实现 SumNumbers 接口的类也是如此。

例 10-9 显示了私有方法在接口中的应用。

例 10-9　测试接口中的私有方法

```
class PrivateDemo implements SumNumbers {} ❶

import org.junit.Test;
import static org.junit.Assert.*;

public class SumNumbersTest {
    private SumNumbers demo = new PrivateDemo();

    @Test
    public void addEvens() throws Exception {
        assertEquals(2 + 4 + 6, demo.addEvens(1, 2, 3, 4, 5, 6)); ❷
    }

    @Test
    public void addOdds() throws Exception {
        assertEquals(1 + 3 + 5, demo.addOdds(1, 2, 3, 4, 5, 6));  ❷
    }
}
```

❶ 实现 SumNumbers 接口的类

❷ 调用委托给私有方法的公共方法

由于只能实例化一个类，程序创建了一个实现 `SumNumbers` 接口的空类。这个名为 `PrivateDemo` 的类被实例化，其公共接口方法可以被调用。

10.3 创建不可变集合

问题

用户希望在 Java 9 中创建不可变列表、集合或映射。

方案

使用 Java 9 新增的静态工厂方法 `List.of`、`Set.of` 与 `Map.of`。

讨论

根据 Javadoc 的描述，可以使用 Java 9 引入的各种 `List.of()` 静态工厂方法方便地创建不可变列表。通过这些方法创建的 `List` 实例具有以下特征。

- 结构上是不可变的（structurally immutable），即无法执行添加、删除或替换元素的操作，且调用任何更改器方法（mutator method）都会抛出 `UnsupportedOperationException`。然而，如果包含的元素本身是可变的，则可能导致 `List` 的内容发生变化。
- 禁止使用 `null` 元素，尝试创建包含 `null` 元素的 `List` 实例将抛出 `NullPointerException`。
- 如果所有元素是可序列化的（serializable），则 `List` 实例也是可序列化的。
- 列表中元素的顺序与所提供参数的顺序相同，与所提供数组中元素的顺序也相同。
- 根据 Serialized Form 页面的规定进行序列化。

例 10-10 列出了 `List.of` 方法所有可用的重载形式。

例 10-10　用于创建不可变列表的静态工厂方法

```
static <E> List<E>    of()
static <E> List<E>    of(E e1)
static <E> List<E>    of(E e1, E e2)
static <E> List<E>    of(E e1, E e2, E e3)
static <E> List<E>    of(E e1, E e2, E e3, E e4)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5, E e6)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
static <E> List<E>    of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9,
    E e10)
static <E> List<E>    of(E... elements)
```

正如 Javadoc 指出的那样，通过上述方法创建的列表在**结构上**是不可变的，因此无法在 `List` 上调用任何常规的更改器方法。换言之，调用 `add`、`addAll`、`clear`、`remove`、`removeAll`、`replaceAll` 以及 `set` 都会抛出 `UnsupportedOperationException`。例 10-11 显示

了部分测试用例。[8]

例 10-11　不可变列表的应用

```
@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityAdd() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.add(99);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityClear() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.clear();
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityRemove() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.remove(0);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityReplace() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.replaceAll(n -> -n);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilitySet() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.set(0, 99);
}
```

然而，如果列表包含的对象本身是可变的，那么列表也可能发生变化。为说明这个问题，我们创建一个名为 `Holder` 的类。这个类很简单，它包含一个可变值 `x`，如例 10-12 所示。

例 10-12　包含可变值的简单类

```
public class Holder {
    private int x;

    public Holder(int x) { this.x = x; }

    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }
}
```

注 8：完整的测试用例请参见本书源代码。

如果创建一个 Holder 实例的不可变列表，由于 Holder 包含的值是可变的，列表也会发生变化，如例 10-13 所示。

例 10-13　修改包装的整数

```
@Test
public void areWeImmutableOrArentWe() throws Exception {
    List<Holder> holders = List.of(new Holder(1), new Holder(2)); ❶
    assertEquals(1, holders.get(0).getX());

    holders.get(0).setX(4);                                        ❷
    assertEquals(4, holders.get(0).getX());
}
```

❶ Holder 实例的不可变列表

❷ 修改 Holder 中包含的值

虽然上述代码可以运行且不违反文档规定，但有悖于开发所应遵循的最佳实践。换言之，如果计划使用不可变列表，应尽量在列表中包含不可变对象。

类似地，根据 Javadoc 的描述，可以使用各种 Set.of() 方法方便地创建不可变集合。通过这些方法创建的 Set 实例具有以下特征。

- 创建时（creation time）拒绝重复元素，传递给静态工厂方法的重复元素会导致 IllegalArgumentException。
- 未指定集合元素的迭代顺序，因此它可能会发生变化。

所有 Set.of() 方法的签名与对应的 List.of() 方法相同，只不过返回的是 Set<E>。

Map.of() 方法同样如此，但其签名传入交替的键和值作为参数，如例 10-14 所示。

例 10-14　用于创建不可变映射的静态工厂方法

```
static <K,V> Map<K,V>    of()
static <K,V> Map<K,V>    of(K k1, V v1)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
    K k9, V v9)
static <K,V> Map<K,V>    of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
    K k9, V v9, K k10, V v10)
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
```

在创建包含最多 10 个条目的映射时，虽然可以使用相应的 Map.of() 方法（交替传入键和值），不过这并非最佳方案，因此 Map 接口还定义了另外两种静态方法，即 ofEntries 和 entry：

```
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
static <K,V> Map.Entry<K,V>     entry(K k, V v)
```

例 10-15 显示了如何利用上面介绍的各种方法创建不可变映射。

例 10-15　利用各种方法创建不可变映射

```
@Test
public void immutableMapFromEntries() throws Exception {
    Map<String, String> jvmLanguages = Map.ofEntries(               ❶
        Map.entry("Java", "http://www.oracle.com/technetwork/java/index.html"),
        Map.entry("Groovy", "http://groovy-lang.org/"),
        Map.entry("Scala", "http://www.scala-lang.org/"),
        Map.entry("Clojure", "https://clojure.org/"),
        Map.entry("Kotlin", "http://kotlinlang.org/"));

    Set<String> names = Set.of("Java", "Scala", "Groovy", "Clojure", "Kotlin");
    List<String> urls = List.of("http://www.oracle.com/technetwork/java/index.html",
            "http://groovy-lang.org/",
            "http://www.scala-lang.org/",
            "https://clojure.org/",
            "http://kotlinlang.org/");

    Set<String> keys = jvmLanguages.keySet();
    Collection<String> values = jvmLanguages.values();

    names.forEach(name -> assertTrue(keys.contains(name)));
    urls.forEach(url -> assertTrue(values.contains(url)));

    Map<String, String> javaMap = Map.of("Java",          ❷
            "http://www.oracle.com/technetwork/java/index.html",
            "Groovy",
            "http://groovy-lang.org/",
            "Scala",
            "http://www.scala-lang.org/",
            "Clojure",
            "https://clojure.org/",
            "Kotlin",
            "http://kotlinlang.org/");
    javaMap.forEach((name, url) -> assertTrue(
            jvmLanguages.keySet().contains(name) && \
              jvmLanguages.values().contains(url)));
}
```

❶ 使用 Map.ofEntries 方法

❷ 使用 Map.of 方法

可以看到，将 ofEntries 与 entry 方法结合在一起使用有助于简化代码。

另见

有关在 Java 8 以及更早版本中创建不可变集合的讨论请参见范例 4.8。

10.4　新增的Stream方法

问题

用户希望使用 Java 9 为 `Stream` 接口添加的新特性。

方案

使用 `Stream` 接口新增的 `ofNullable`、`iterate`、`takeWhile` 以及 `dropWhile` 方法。

讨论

Java 9 为 `Stream` 接口引入了 `ofNullable`、`iterate`、`takeWhile`、`dropWhile` 等新方法，本范例将讨论它们的用法。

1. ofNullable方法

在 Java 8 中，`of` 方法包括两种形式，一种传入单个值，另一种传入可变参数列表。无论哪种形式，参数都不能为空。

而在 Java 9 中，新的 `ofNullable` 方法可以在参数不为空时返回一个包装值的单元素流，为空时返回一个空流。例 10-16 的用例显示了该方法的应用。

例 10-16　`ofNullable` 方法的应用

```
@Test
public void ofNullable() throws Exception {
    Stream<String> stream = Stream.ofNullable("abc");  ❶
    assertEquals(1, stream.count());

    stream = Stream.ofNullable(null);                  ❷
    assertEquals(0, stream.count());
}
```

❶ 单元素流

❷ 返回 `Stream.empty()`

在本例中，`count` 方法返回流中非空元素的数量，我们可以借此在任何参数上使用 `ofNullable` 方法，而无须首先检查参数是否为空。

2. 传入Predicate的iterate方法

另一种有趣的方法是 `iterate`。在 Java 8 中，`iterate` 方法的签名如下：

```
static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

创建流时，从初始元素种子开始，对种子依次应用一元运算符以产生后续元素。由于生成

的流是一个无限流，通常需要采用 `limit` 或其他短路操作（short-circuiting operation，如 `findFirst` 或 `findAny`）来控制返回流的大小。

在 Java 9 中，`iterate` 方法新增了一种重载形式，它传入 `Predicate` 作为第二个参数：

```
static<T> Stream<T> iterate(T seed, Predicate<? super T> hasNext,
    UnaryOperator<T> next)
```

创建流时，从初始元素种子开始，对种子应用一元运算符，直至值不再满足谓词 `hasNext`。

相关应用如例 10-17 所示。

例 10-17　传入 `Predicate` 的 `iterate` 方法

```
@Test
public void iterate() throws Exception {
    List<BigDecimal> bigDecimals =                  ❶
            Stream.iterate(BigDecimal.ZERO, bd -> bd.add(BigDecimal.ONE))
            .limit(10)
            .collect(Collectors.toList());

    assertEquals(10, bigDecimals.size());

    bigDecimals = Stream.iterate(BigDecimal.ZERO, ❷
            bd -> bd.longValue() < 10L,
            bd -> bd.add(BigDecimal.ONE))
            .collect(Collectors.toList());

    assertEquals(10, bigDecimals.size());
}
```

❶ 创建 `BigDecimal` 流（Java 8 实现）

❷ 创建 `BigDecimal` 流（Java 9 实现）

第一个流使用 `iterate` 方法，并通过 `limit` 方法控制大小，这是 Java 8 的实现方式；第二个流传入 `Predicate` 作为第二个参数，看起来更像是传统的 `for` 循环。

3. takeWhile与dropWhile方法

Java 9 新增了 `takeWhile` 和 `dropWhile` 方法，二者基于谓词获取流的某一部分。根据 Javadoc 的描述，对于有序流，`takeWhile` 方法从流的起始位置开始，返回“匹配给定谓词的元素的最长前缀”。

`dropWhile` 方法的作用正好相反，在丢弃匹配给定谓词的元素的最长前缀后，该方法将返回流的其余元素。

两种方法在有序流上的应用如例 10-18 所示。

例 10-18　获取与丢弃流中的元素

```
@Test
public void takeWhile() throws Exception {
    List<String> strings = Stream.of("this is a list of strings".split(" "))
            .takeWhile(s -> !s.equals("of"))  ❶
            .collect(Collectors.toList());
```

```
        List<String> correct = Arrays.asList("this", "is", "a", "list");
        assertEquals(correct, strings);
    }

    @Test
    public void dropWhile() throws Exception {
        List<String> strings = Stream.of("this is a list of strings".split(" "))
                .dropWhile(s -> !s.equals("of"))  ❷
                .collect(Collectors.toList());
        List<String> correct = Arrays.asList("of", "strings");
        assertEquals(correct, strings);
    }
```

❶ 当不再满足谓词时，返回谓词之前的元素

❷ 当不再满足谓词时，返回谓词之后的元素

可以看到，两种方法在同一个位置将流拆分，不过 `takeWhile` 返回拆分位置之前的元素，而 `dropWhile` 返回拆分位置之后的元素。

`takeWhile` 方法的最大优点在于它是一种短路操作：对于一个包含大量排序元素的集合，只要达到所设定的条件，就可以停止求值。

例如，假设存在一个由客户订单构成的集合，集合中的元素以降序方式排序。借由 `takeWhile` 方法，我们可以只获取高于某个阈值的订单，而不必筛选每个元素。

例 10-19 模拟了这种情况。程序生成 50 个 0 到 100 之间的随机整数，对它们做降序排序，然后仅返回大于 90 的整数。

例 10-19　对整数流应用 `takeWhile` 方法

```
Random random = new Random();
List<Integer> nums = random.ints(50, 0, 100) ❶
        .boxed()                             ❷
        .sorted(Comparator.reverseOrder())
        .takeWhile(n -> n > 70)              ❸
        .collect(Collectors.toList());
```

❶ 生成 50 个 0 到 100 之间的随机整数

❷ 将这些整数装箱，以便采用 `Comparator` 排序并收集

❸ 将流拆分并返回大于 90 的整数

改用 `dropWhile` 方法或许能让本例看起来更为直观（尽管效率未必会提高），如例 10-20 所示。

例 10-20　对整数流应用 `dropWhile` 方法

```
Random random = new Random();
List<Integer> nums = random.ints(50, 0, 100)
        .sorted()              ❶
        .dropWhile(n -> n < 90)  ❷
        .boxed()
        .collect(Collectors.toList());
```

❶ 升序排序

❷ 将流拆分并返回大于 90 的整数

类似 `takeWhile` 和 `dropWhile` 这样的方法在其他语言中已存在多年，Java 9 将二者正式引入 Java。

10.5 下游收集器：filtering与flatMapping

问题

用户希望将元素作为下游收集器（downstream collector）的一部分进行筛选，或将集合的集合展平。

方案

使用 Java 9 为 `Collectors` 类新增的 `filtering` 和 `flatMapping` 方法。

讨论

Java 8 为 `Collectors` 类引入了 `groupingBy` 操作，用于根据特定的属性将对象分组。分组操作将产生一个“键 – 值列表”映射（`Map<K, List<T>>`）。Java 8 还支持使用**下游**收集器，可以不必生成列表，而是对列表进行后期处理以获取其大小，或将列表映射为其他内容。

Java 9 新增了两种下游收集器，它们是 `filtering` 和 `flatMapping`。

1. filtering方法

假设存在两个类，一个类是 `Task`，该类包括描述预算的属性以及承担任务的开发人员列表；另一个类是 `Developer`，它的实例用于描述开发人员。两个类如例 10-21 所示。

例 10-21　Task 和 Developer 类

```
public class Task {
    private String name;
    private long budget;
    private List<Developer> developers = new ArrayList<>();

    // 构造函数、getter、setter等
}

public class Developer {
    private String name;

    // 构造函数、getter、setter等
}
```

首先，我们根据预算对任务进行分组。例 10-22 显示了一个简单的 `groupingBy` 操作。

例 10-22　根据预算对任务分组

```
Developer venkat = new Developer("Venkat");
Developer daniel = new Developer("Daniel");
Developer brian = new Developer("Brian");
```

```
Developer matt = new Developer("Matt");
Developer nate = new Developer("Nate");
Developer craig = new Developer("Craig");
Developer ken = new Developer("Ken");

Task java = new Task("Java stuff", 100);
Task altJvm = new Task("Groovy/Kotlin/Scala/Clojure", 50);
Task javaScript = new Task("JavaScript (sorry)", 100);
Task spring = new Task("Spring", 50);
Task jpa = new Task("JPA/Hibernate", 20);

java.addDevelopers(venkat, daniel, brian, ken);
javaScript.addDevelopers(venkat, nate);
spring.addDevelopers(craig, matt, nate, ken);
altJvm.addDevelopers(venkat, daniel, ken);

List<Task> tasks = Arrays.asList(java, altJvm, javaScript, spring, jpa);

Map<Long, List<Task>> taskMap = tasks.stream()
        .collect(groupingBy(Task::getBudget));
```

由此建立了预算金额与分配该预算的任务列表之间的映射：

```
 50: [Groovy/Kotlin/Scala/Clojure, Spring]
 20: [JPA/Hibernate]
100: [Java stuff, JavaScript (sorry)]
```

如果只希望获取预算超过某个阈值的任务，可以添加一个 `filter` 操作，如例 10-23 所示。

例 10-23　利用 `filter` 操作进行分组

```
taskMap = tasks.stream()
        .filter(task -> task.getBudget() >= THRESHOLD)
        .collect(groupingBy(Task::getBudget));
```

如果阈值为 50，程序的输出如下：

```
 50: [Groovy/Kotlin/Scala/Clojure, Spring]
100: [Java stuff, JavaScript (sorry)]
```

可以看到，预算低于阈值的任务不会出现在输出映射中，不过仍然有办法显示这些任务：在 Java 9 中，`Collectors` 类新增了一个名为 `filtering` 的静态方法，它与 `filter` 类似，只不过用于下游任务列表的筛选。`filtering` 方法的用法如例 10-24 所示。

例 10-24　利用下游筛选器进行分组

```
taskMap = tasks.stream()
    .collect(groupingBy(Task::getBudget,
         filtering(task -> task.getBudget() >= 50, toList())));
```

此时，所有预算金额都会以键的形式显示出来，但预算低于阈值的任务不会出现在列表值中：

```
 50: [Groovy/Kotlin/Scala/Clojure, Spring]
 20: []
100: [Java stuff, JavaScript (sorry)]
```

因此，`filtering` 操作是一种下游收集器，可以对分组操作产生的列表操作。

2. flatMapping方法

那么，如何获取承担每项任务的开发人员列表呢？如例 10-25 所示，借由基本的分组操作，可以根据任务名对任务分组。

例 10-25　根据任务名分组

```
Map<String, List<Task>> tasksByName = tasks.stream()
        .collect(groupingBy(Task::getName));
```

（格式化后的）输出如下：

```
                 Java stuff: [Java stuff]
Groovy/Kotlin/Scala/Clojure: [Groovy/Kotlin/Scala/Clojure]
         JavaScript (sorry): [JavaScript (sorry)]
                     Spring: [Spring]
              JPA/Hibernate: [JPA/Hibernate]
```

为获取与任务关联的开发人员列表，我们使用下游收集器 `mapping`，如例 10-26 所示。

例 10-26　承担每项任务的开发人员列表

```
Map<String, Set<List<Developer>>> map = tasks.stream()
        .collect(groupingBy(Task::getName,
              Collectors.mapping(Task::getDevelopers, toSet())));
```

不过，返回类型是 `Set<List<Developer>>`，而我们需要的是一个下游 `flatMap` 操作来展平集合的集合。为此，可以使用 `Collectors` 类新增的 `flatMapping` 方法，如例 10-27 所示。

例 10-27　利用 flatMapping 方法获取一组开发人员

```
Map<String, Set<Developer>> task2setdevs = tasks.stream()
        .collect(groupingBy(Task::getName,
                Collectors.flatMapping(task -> task.getDevelopers().stream(),
                toSet())));
```

（格式化后的）输出如下：

```
                 Java stuff: [Daniel, Brian, Ken, Venkat]
Groovy/Kotlin/Scala/Clojure: [Daniel, Ken, Venkat]
         JavaScript (sorry): [Nate, Venkat]
                     Spring: [Craig, Ken, Matt, Nate]
              JPA/Hibernate: []
```

`Collectors.flatMapping` 方法类似于 `Stream.flatMap` 方法。需要注意的是，`flatMapping` 方法的第一个参数应是一个流，它可以为空，或不依赖于数据源。

另见

有关下游收集器的讨论请参见范例 4.6，有关 `flatMap` 操作的讨论请参见范例 3.11。

10.6 新增的Optional方法

问题

用户希望执行以下操作：将 Optional 映射并展平到包含元素的流中；从几种可能的条件中进行选择；当元素存在时进行某种操作，否则返回默认值。

方案

使用 Java 9 为 Optional 类新增的 stream、or 或 ifPresentOrElse 方法。

讨论

Java 8 引入了 Optional 类，用于告知客户端返回值可能合法为 null。返回的并非 null，而是空 Optional。对可能返回（或不返回）值的方法而言，Optional 是一种不错的包装器。

1. stream方法

例 10-28 显示了一种根据 ID 检索客户的查找器方法（finder method）。

例 10-28　根据 ID 查找客户

```
public Optional<Customer> findById(int id) {
    return Optional.ofNullable(map.get(id));
}
```

findById 方法假设客户包含在内存的 Map 中。map.get 方法在键存在时返回一个值，否则返回 null，因此将其作为 Optional.ofNullable 方法的参数时，要么在 Optional 中包装一个非空值，要么返回一个空 Optional。



由于 Optional.of 方法在参数为 null 时将抛出异常，采用 Optional.ofNullable(arg) 更方便，其实现为 arg != null ? Optional.of(arg) : Optional.empty()。

findById 方法返回的是 Optional<Customer>，如果尝试返回客户集合则略显复杂。在 Java 8 中，不难写出如例 10-29 所示的代码。

例 10-29　在 Optional 上使用 filter 和 map 方法

```
public Collection<Customer> findAllById(Integer... ids) {
    return Arrays.stream(ids)
            .map(this::findById)          ❶
            .filter(Optional::isPresent)  ❷
            .map(Optional::get)           ❸
            .collect(Collectors.toList());
}
```

❶ 映射到 Stream<Optional<Customer>>

❷ 筛掉所有空 Optional

❸ 调用 get 方法以映射到 Stream<Customer>

上述实现并不难，但利用 Java 9 为 Optional 类新增的 stream 方法，可以使这个过程更简单。stream 方法的签名如下：

```
Stream<T> stream()
```

如果值存在，stream 方法将返回一个顺序的单元素流，流中仅包含该值；如果值不存在，stream 方法将返回一个空流。借由 stream 方法，可选的客户流可以直接转换为客户流，如例 10-30 所示。

例 10-30　使用传入 Optional.stream 的 flatMap 方法

```
public Collection<Customer> findAllById(Integer... ids) {
    return Arrays.stream(ids)
            .map(this::findById)           ❶
            .flatMap(Optional::stream)     ❷
            .collect(Collectors.toList());
}
```

❶ 映射到 Stream<Optional<Customer>>

❷ 映射并展平为 Stream<Customer>

使用 stream 纯粹是为了方便，但它不失为一种有用的方法。

2. or方法

orElse 方法用于从 Optional 中提取值，它传入默认值作为参数：

```
Customer customer = findById(id).orElse(Customer.DEFAULT)
```

也可以使用传入 Supplier 来创建默认值的 orElseGet 方法，不过这是一种成本较高的操作：

```
Customer customer = findById(id).orElseGet(() -> createDefaultCustomer())
```

orElse 和 orElseGet 方法均返回 Customer 实例。而 Java 9 新增的 or 方法可以在给定 Supplier 时返回 Optional<Customer>，从而将查找客户的其他方式链接在一起。

or 方法的签名如下：

```
Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)
```

如果值存在，or 方法将返回描述该值的 Optional，否则返回由 Supplier 生成的 Optional。

我们现在可以通过多种方式查找客户，例 10-31 展示了 or 方法的应用。

例 10-31　改用 or 方法

```
public Optional<Customer> findById(int id) {
    return findByIdLocal(id)
            .or(() -> findByIdRemote(id))
            .or(() -> Optional.of(Customer.DEFAULT));
}
```

程序首先在本地缓存中搜索客户，再访问远程服务器。如果两种方式都未能找到非空 Optional，最后一个子句创建一个默认值，将其包装在 Optional 中并返回。

3. ifPresentOrElse方法

如果 Optional 不为空，ifPresent 方法将执行 Consumer 指定的操作，如例 10-32 所示。

例 10-32　利用 ifPresent 方法仅打印非空客户

```
public void printCustomer(Integer id) {
    findByIdLocal(id).ifPresent(System.out::println);  ❶
}

public void printCustomers(Integer... ids) {
    Arrays.asList(ids)
            .forEach(this::printCustomer);
}
```

❶ 仅打印非空 Optional

虽然 ifPresent 方法很有用，不过如果返回的 Optional 为空，我们可能还希望执行其他操作。Java 9 新增的 ifPresentOrElse 方法包括两个参数，在 Optional 为空时将执行第二个参数 Runnable 指定的操作。该方法的签名如下：

```
void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
```

使用 ifPresentOrElse 方法时，只需提供一个不传入任何参数并返回 void 的 lambda 表达式，如例 10-33 所示。

例 10-33　打印客户或默认消息

```
public void printCustomer(Integer id) {
    findByIdLocal(id).ifPresentOrElse(System.out::println,
            () -> System.out.println("Customer with id=" + id + " not found"));
}
```

如果找到指定的客户，程序将打印该客户，否则打印默认消息。

Optional 类新增的这些方法均未从根本上改变各自的用途，但它们确实为开发提供了更大的便利。

另见

有关 Java 8 引入的 Optional 类请参见第 6 章。

10.7　日期范围

问题

用户希望返回两个给定端点之间的日期流。

方案

使用 Java 9 为 LocalDate 类新增的 datesUntil 方法。

讨论

较之 `java.util.Date`、`java.util.Calendar` 以及 `java.sql.Timestamp` 类，Java 8 引入的 Date-Time API 是一种巨大改进。而 Java 9 新增的 `datesUntil` 方法解决了 Date-Time API 中一个令人头疼的问题：难以方便地创建日期流。

在 Java 8 中，创建日期流最简单的方式是以初始日期为基准，再添加一个偏移量。例如，为返回相隔一周的两个给定端点之间的所有天数，我们可能会写出如例 10-34 所示的代码。

例 10-34　返回两个日期之间的天数（存在问题）

```
public List<LocalDate> getDays_java8(LocalDate start, LocalDate end) {
    Period period = start.until(end);
    return IntStream.range(0, period.getDays())  ❶
            .mapToObj(start:plusDays)
            .collect(Collectors.toList());
}
```

❶ 实为陷阱！正确实现参见例 10-35。

程序首先计算两个日期之间的 `Period`，然后在二者之间创建一个 `IntStream`。执行程序，观察结束日期和开始日期相隔一周时的情况：

```
LocalDate start = LocalDate.of(2017, Month.JUNE, 10);
LocalDate end = LocalDate.of(2017, Month.JUNE, 17);
System.out.println(dateRange.getDays_java8(start, end));

// [2017-06-10, 2017-06-11, 2017-06-12, 2017-06-13,
// 2017-06-14, 2017-06-15, 2017-06-16]
```

上述代码看似正确，实则有误。如果将结束日期改为与开始日期相隔正好一个月，很容易就能看出问题所在：

```
LocalDate start = LocalDate.of(2017, Month.JUNE, 10);
LocalDate end = LocalDate.of(2017, Month.JULY, 10);
System.out.println(dateRange.getDays_java8(start, end));

// []
```

可以看到，程序没有返回任何值。原因在于 `period.getDays` 方法返回的只是两个天数字段之间的天数，而非两个日期之间的总天数（`getMonths`、`getYears` 等方法同样如此）。如上所示，由于开始日期和结束日期的天数相同，虽然月份不同，结果仍然是一个大小为 0 的范围。

为解决这个问题，应采用实现 `TemporalUnit` 接口的 `ChronoUnit` 枚举，它定义了 `DAYS`、`MONTHS` 等多个枚举常量。Java 8 的正确实现如例 10-35 所示。

例 10-35　返回两个日期之间的天数（正确实现）

```
public List<LocalDate> getDays_java8(LocalDate start, LocalDate end) {
    Period period = start.until(end);
    return LongStream.range(0, ChronoUnit.DAYS.between(start, end)) ❶
            .mapToObj(start:plusDays)
            .collect(Collectors.toList());
}
```

❶ 正确无误

我们也可以使用 `iterate` 方法，但需要了解两个日期之间的天数，如例 10-36 所示。

例 10-36　`LocalDate` 的迭代

```
public List<LocalDate> getDaysByIterate(LocalDate start, int days) {
    return Stream.iterate(start, date -> date.plusDays(1))
            .limit(days)
            .collect(Collectors.toList());
}
```

好在 Java 9 引入的新方法使问题得以简化。`LocalDate` 类新增了一种名为 `datesUntil` 的方法，其重载形式传入 `Period` 作为参数。`datesUntil` 方法的签名如下：

```
Stream<LocalDate> datesUntil(LocalDate endExclusive)
Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)
```

不传入 `Period` 的 `datesUntil` 方法实际上相当于将日期增量设置为一天，即 `datesUntil(endExclusive)` 等效于 `datesUntil(endExclusive, Period.ofDays(1))`。

采用 `datesUntil` 方法返回两个日期之间的天数要简单得多，如例 10-37 所示。

例 10-37　返回两个日期之间的天数（Java 9 实现）

```
public List<LocalDate> getDays_java9(LocalDate start, LocalDate end) {
    return start.datesUntil(end)                          ❶
            .collect(Collectors.toList());
}

public List<LocalDate> getMonths_java9(LocalDate start, LocalDate end) {
    return start.datesUntil(end, Period.ofMonths(1)) ❷
            .collect(Collectors.toList());
}
```

❶ 相当于 `Period.ofDays(1)`

❷ 日期增量为一个月

我们可以使用所有常规的流处理技术对 `datesUntil` 方法产生的 `Stream` 操作。

另见

有关在 Java 8 中计算两个日期之间的天数，请参见范例 8.8。

附录A

泛型与Java 8

A.1 背景

Java 1.5 引入了泛型（generics）的概念。遗憾的是，大部分 Java 开发人员对于泛型的了解仅停留在完成工作所需的层面上。随着 Java 8 的兴起，Javadoc 中出现了不少使用泛型的方法签名。以 `java.util.Map.Entry` 接口的 `comparingByKey` 方法为例：

```
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>>
    comparingByKey()
```

以及 `java.util.Comparator` 接口的 `comparing` 方法：

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(
    Function<? super T,? extends U> keyExtractor)
```

甚至 `java.util.stream.Collectors` 类的 `groupingBy` 方法：

```
static <T,K,D,A,M extends Map<K, D>> Collector<T,?,M> groupingBy(
    Function<? super T,? extends K> classifier,  Supplier<M> mapFactory,
    Collector<? super T,A,D> downstream)
```

显然，对泛型仅有最低限度的了解是远远不够的。本附录旨在分析这些签名的结构，以帮助读者在开发中更有效地应用 API。

A.2 众所周知的事实

在使用 `List` 或 `Set` 这样的集合时，可以将元素的类名置于尖括号中，以声明所包含元素的类型：

```
List<String> strings = new ArrayList<String>();
Set<Employee> employees = new HashSet<Employee>();
```



通过在之后的示例代码中引入**钻石运算符**（diamond operator），Java 7 能在一定程度上简化语法。由于等号左侧的引用已经声明了集合以及所包含的类型（如 `List<String>` 或 `List<Integer>`），等号右侧的实例化无须再次声明。我们可以将其简写为 `new ArrayList<>()`，而不必将类型置于尖括号中。

声明集合的数据类型可以实现两个目的：

- 能避免不慎将错误的类型置于集合中
- 无须再将检索到的值强制转换为合适的类型

如例 A-1 所示，在声明 `strings` 变量之后，就只能向集合添加 `String` 实例，并在检索到某项时自动获得一个 `String`。

例 A-1　简单的泛型示例

```
List<String> strings = new ArrayList<>();
strings.add("Hello");
strings.add("World");
// strings.add(new Date());    ❶
// Integer i = strings.get(0); ❶

for (String s : strings) {     ❷
    System.out.printf("%s has length %d%n", s, s.length());
}
```

❶ 无法编译

❷ for-each 循环了解所包含的数据类型为 `String`

对插入过程应用类型安全（type safety）很方便，但开发人员很少会犯这个错误。不过，如果不必首先强制转换就可以处理检索类型，能极大简化代码。[1]

另一个众所周知的事实是，无法为泛型集合添加基本数据类型（primitive type）。换言之，目前尚无法定义 `List<int>` 或 `List<double>`。[2] 幸运的是，Java 1.5 在引入泛型的同时也引入了自动装箱和拆箱。因此，如果希望在泛型类型（generic type）中储存基本数据类型，可以通过包装类（wrapper class）声明该类型，如例 A-2 所示。

例 A-2　在泛型集合中使用基本数据类型

```
List<Integer> ints = new ArrayList<>();
ints.add(3); ints.add(1); ints.add(4);
ints.add(1); ints.add(9); ints.add(2);
System.out.println(ints);

for (int i : ints) {
    System.out.println(i);
}
```

注 1：在整个职业生涯中，我从未不慎将错误的类型添加到列表中。不过即便只是考虑到糟糕的语法，去掉强制转换过程也是值得的。

注 2：Java 10（Valhalla 项目）已提出将基本数据类型添加到集合中。

可以看到，Java 在插入时将 int 值包装在 Integer 实例中，并在检索时从 Integer 实例中取出这些值。尽管装箱和拆箱的效率有待商榷，但代码确实很容易编写。

此外，Java 开发人员耳熟能详的一点是，如果一个类使用了泛型，那么类型本身采用尖括号中的大写字母表示。例如，Javadoc 对 java.util.List 接口的描述如下：

```
public interface List<E> extends Collection<E>
```

其中 E 是类型参数（type parameter），且接口中的方法使用相同的类型参数。例 A-3 显示了 List 接口声明的部分方法。

例 A-3　List 接口声明的部分方法

```
boolean add(E e)                              ❶
boolean addAll(Collection<? extends E> c)     ❷
void    clear()                               ❸
boolean contains(Object o)                    ❸
boolean containsAll(Collection<?> c)          ❹
E       get(int index)                        ❶
```

❶ 类型参数 E 用作参数或返回类型

❷ **有界**通配符

❸ 与类型本身无关的方法

❹ **未知**类型

可以看到，某些方法使用声明的泛型类型 E 作为参数或返回类型，某些方法（特别是 clear 和 contains）完全不使用类型，还有部分方法使用问号作为通配符。

请注意，在非泛型类中声明泛型方法是合法的。这种情况下，泛型参数被声明为方法签名的一部分。以工具类 java.util.Collections 为例，它定义了以下静态方法：

```
static <T>   List<T>    emptyList()
static <K,V> Map<K,V>   emptyMap()
static <T>   boolean    addAll(Collection<? super T> c, T... elements)
static <T extends Object & Comparable<? super T>>
    T min(Collection<? extends T> coll)
```

如上所示，emptyList、addAll、min 这三种方法声明了泛型参数 T。emptyList 方法通过 T 来指定 List 中包含的类型，而 emptyMap 方法在泛型映射中使用 K 和 V 来表示键的类与值的类。

addAll 方法声明了泛型类型 T，并使用 Collection<? super T> c 作为方法的第一个参数，T 类型的可变参数列表作为第二个参数。? super T 是一种有界通配符（bounded wildcard），稍后将对此做讨论。

从 min 方法可以看出泛型类型是如何提供安全性的，但其签名结构或许不那么一目了然。后面将详细讨论该方法的签名，目前不妨这样理解：T 是有界的，它既是 Object 的子类，又实现了 Comparable 接口，其中 Comparable 定义为 T 或 T 的任何父类。min 方法的参数与 T 或 T 的任何子类的 Collection 有关。

最后，通配符将众所周知的语法以我们不那么熟悉的形式表现出来。例如，某些语法看似继承，但实际上根本不是。

A.3 容易忽略的事实

许多开发人员或许惊讶于 ArrayList<String> 和 ArrayList<Object> 并无实质性的关联。如例 A-4 所示，可以将 Object 的子类添加到 Object 集合中。

例 A-4　List<Object> 的应用

```
List<Object> objects = new ArrayList<Object>();
objects.add("Hello");
objects.add(LocalDate.now());
objects.add(3);
System.out.println(objects);
```

很好！由于 String 是 Object 的子类，可以将 String 引用赋给 Object 引用。读者可能认为，在声明字符串列表之后就能为其添加对象，但实际情况并非如此，如例 A-5 所示。

例 A-5　List<String> 与对象一起使用

```
List<String> strings = new ArrayList<>();
String s = "abc";
Object o = s;                          ❶
// strings.add(o);                     ❷

// List<Object> moreObjects = strings; ❸
// moreObjects.add(new Date());
// String s = moreObjects.get(0);      ❹
```

❶ 合法

❷ 不合法

❸ 同样不合法，但假设其合法

❹ 损坏的集合

由于 String 是 Object 的子类，我们可以将 String 引用赋给 Object 引用，但无法将 Object 引用添加到 List<String>。这似乎有些奇怪，原因在于 List<String> **并非** List<Object> 的子类。在声明类型时，可以添加的**唯一**实例就是所声明的类型，使用子类或超类实例均不合法。换言之，参数化类型（parameterized type）具有**不变性**（invariance）。

在本例中，从注释掉的语句不难看出为何 List<String> 不是 List<Object> 的子类。假设可以将 List<String> 赋给 List<Object>，那么通过对象引用列表就能将非字符串的内容添加到列表中。这样一来，采用字符串列表的原始引用检索时会导致强制转换异常，编译器将无法判断转换是否有效。

不过，如果定义了一个数字列表，应该就可以为列表添加整数、浮点数与双精度浮点数。为此，我们需要在类型边界（type bound）中使用通配符。

A.4　通配符与PECS

通配符是一种使用问号（?）的类型参数，可能存在（也可能不存在）上界或下界。

A.4.1　无界通配符

没有边界的类型参数很有用，不过也存在一定局限性。如例 A-6 所示，对一个声明为无界类型的 List 而言，可以读取，但无法写入。

例 A-6　使用无界通配符的 List

```
List<?> stuff = new ArrayList<>();
// stuff.add("abc");           ❶
// stuff.add(new Object());
// stuff.add(3);
int numElements = stuff.size(); ❷
```

❶ 不允许进行添加操作

❷ numElements 为 0

由于无法传入任何内容，上述代码的意义不大。不过，无界 List 的一种用途在于，所有传入 List<?> 作为参数的方法都会在调用时接受任何列表，如例 A-7 所示。

例 A-7　无界 List 作为方法参数

```
private static void printList(List<?> list) {
    System.out.println(list);
}

public static void main(String[] args) {
    // 创建列表ints、strings与stuff
    printList(ints);
    printList(strings);
    printList(stuff);
}
```

读者或许还记得 List<E> 接口声明的 containsAll 方法（例 A-3）：

```
boolean containsAll(Collection<?> c)
```

只有当前列表包含指定集合的所有元素时，containsAll 方法才返回 true。由于方法参数使用的是无界通配符，实现仅限于以下两类方法：

- Collection 接口定义的、不需要包含类型的方法
- Object 类定义的方法

对于 containsAll 方法，上述条件完全符合。引用实现中的默认实现（AbstractCollection 类）通过 iterator 方法遍历参数，并调用 contains 方法检查其中的所有元素是否也在原始列表中。iterator 和 contains 方法定义在 Collection 接口中，而 equals 方法定义在 Object 类中。contains 实现委托给 Object 类的 equals 和 hashCode 方法，它们可能已经在包含的类型中被重写。就 containsAll 方法而言，它需要的所有方法都是可用的，因此无界通配符不会对该方法的使用造成影响。

问号是设置类型边界的利器，其用法相当多样化。

A.4.2　上界通配符

上界通配符（upper bounded wildcard）使用关键字 `extends` 来设置超类限制。例 A-8 定义了一个支持 `int`、`long`、`double` 甚至 `BigDecimal` 实例的数字列表。



即便采用接口（而不是类）作为上界，也可以使用关键字 `extends`，如 `List<? extends Comparable>`。

例 A-8　具有上界的 List

```
List<? extends Number> numbers = new ArrayList<>();
//        numbers.add(3);                    ❶
//        numbers.add(3.14159);
//        numbers.add(new BigDecimal("3"));
```

❶ 仍然无法添加值

上述代码看似不错，不过虽然可以使用上界通配符定义列表，但仍然无法为列表添加值。原因在于检索值时，编译器并不清楚列表的类型，只知道它继承了 `Number`。

尽管如此，我们可以定义一个传入 `List<? extends Number>` 的方法参数，然后通过不同的列表类型调用方法，如例 A-9 所示。

例 A-9　上界的应用

```
private static double sumList(List<? extends Number> list) {
    return list.stream()
               .mapToDouble(Number::doubleValue)
               .sum();
}

public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
    List<Double> doubles = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
    List<BigDecimal> bigDecimals = Arrays.asList(
        new BigDecimal("1.0"),
        new BigDecimal("2.0"),
        new BigDecimal("3.0"),
        new BigDecimal("4.0"),
        new BigDecimal("5.0")
    );

    System.out.printf("ints sum is         %s%n", sumList(ints));
    System.out.printf("doubles sum is      %s%n", sumList(doubles));
    System.out.printf("big decimals sum is %s%n", sumList(bigDecimals));
}
```

可以看到，使用相应的 `double` 值对 `BigDecimal` 实例求和，会抵消首先使用 `BigDecimal` 所带来的好处，但只有基本类型流 `IntStream`、`LongStream` 与 `DoubleStream` 包括 `sum` 方法。不过这也说明，可以使用 `Number` 的任何子类型（subtype）的列表来调用方法。由于 `Number` 定义了 `doubleValue` 方法，代码成功编译并运行。

从具有上界的列表中访问某个元素时，结果肯定可以被赋给上界类型的引用，如例 A-10 所示。

例 A-10　从上界引用中提取值

```
private static double sumList(List<? extends Number> list) {
    Number num = list.get(0);
    // 其余代码与例A-9相同
}
```

调用方法时，列表元素要么是 `Number`，要么是它的某个子类，因此 `Number` 引用总是正确的。

A.4.3　下界通配符

下界通配符（lower bounded wildcard）表示类的任何父类均满足条件，关键字 `super` 和通配符用于指定下界。以 `List<? super Number>` 为例，引用既可以代表 `List<Number>`，也可以代表 `List<Object>`。

我们通过上界指定变量必须符合的类型，以便方法实现能正常工作。对数字求和时，需要确保变量有一个定义在 `Number` 中的 `doubleValue` 方法。通过直接或重写的形式，`Number` 的所有子类也会包含 `doubleValue` 方法，这就是将输入类型指定为 `List<? extends Number>` 的原因。

而在下界通配符中，我们从列表中取出项目，并添加到不同的集合。目标集合既可以是 `List<Number>`，也可以是 `List<Object>`，因为单个 `Object` 引用可以被赋给一个 `Number`。

接下来，我们将讨论一个经常被引用的示例。尽管它并不符合真正的 Java 8 习惯用法（稍后将解释原因），但的确阐释了下界通配符的概念。

如例 A-11 所示，`numsUpTo` 方法传入两个参数，一个是整数，另一个是列表。采用所有数字填充列表，直至达到第一个参数指定的数字。

例 A-11　`numsUpTo` 方法用于填充给定列表

```
public void numsUpTo(Integer num, List<? super Integer> output) {
    IntStream.rangeClosed(1, num)
             .forEach(output::add);
}
```

`numsUpTo` 方法之所以不符合 Java 8 的习惯用法，是因为它使用提供的列表作为输出变量。这实际上会带来副作用，因此不鼓励使用。尽管如此，通过将第二个参数的类型设置为 `List<? super Integer>`，提供的列表就可以是 `List<Integer>`、`List<Number>` 甚至 `List<Object>` 类型，如例 A-12 所示。

例 A-12　`numsUpTo` 方法的应用

```
ArrayList<Integer> integerList = new ArrayList<>();
ArrayList<Number> numberList = new ArrayList<>();
ArrayList<Object> objectList = new ArrayList<>();

numsUpTo(5, integerList);
numsUpTo(5, numberList);
numsUpTo(5, objectList);
```

所有返回的列表均包含数字 1 到 5。使用下界通配符意味着列表将用于存储整数，但我们可以在任何超类型（supertype）的列表中使用引用。

在上界列表中，我们从列表中提取并使用值；在下界列表中，我们为列表提供值。二者的综合应用构成了所谓的 PECS 原则。

A.4.4　PECS原则

PECS 是“Producer Extends, Consumer Super”的缩写，这是 Joshua Bloch 在 *Effective Java* 一书[3]中引入的一个略显奇怪的术语，但有助于理解泛型的用法。换言之，参数化类型代表生产者（producer）则使用 `extends`，代表消费者（consumer）则使用 `super`。如果参数同时代表生产者和消费者则无须使用通配符，因为满足这两项要求的唯一类型就是显式类型（explicit type）自身。

可以将 PECS 原则归纳如下：

- 仅从数据结构**获取**值时，使用 `extends`；
- 仅向数据结构**写入**值时，使用 `super`；
- 如果需要同时获取和写入值，使用显式类型。

对于本节讨论的某些概念，均有描述这些概念的正式术语，它们经常在 Scala 这样的语言中使用。

术语**协变**（covariance）表示可以使用比原始指定的派生类型更大的类型。在 Java 中，由于 `String[]` 是 `Object[]` 的子类型，数组是协变的；除非使用关键字 `extends` 和通配符，否则集合不是协变的。

术语**逆变**（contravariance）表示可以使用比原始指定的派生类型更小的类型。在 Java 中，通过关键字 `super` 和通配符引入逆变。

术语**不变性**（invariance）表示只能使用原始指定的类型。除非使用 `extends` 或 `super`，否则 Java 中的所有参数化类型都具有不变性。换言之，如果某个方法要求 `List<Employee>`，就必须提供 `List<Employee>`，而不能提供 `List<Object>` 或 `List<Salaried>`[4]。

PECS 是对形式规则（formal rule）的一种重述，即类型构造函数在输入类型中是逆变的，在输出类型中是协变的。某些情况下，也可以将 PECS 原则表述为“读取时使用 `extends`，写入时使用 `super`”（be liberal in what you accept and conservative in what you produce）。

A.4.5　多重边界

在讨论 Java 8 API 中的示例之前，我们先来介绍多重边界（multiple bound）。类型参数可以有多重边界，边界之间通过“&”符号隔开：

注 3：公认的经典 Java 教程，总结了 Java 程序设计中大量极具实用价值的规则，这些规则涵盖了开发中可能遇到的各种问题。——译者注

注 4：协变、逆变与不变性的定义如下。如果 X 和 Y 表示类型，$\leqslant$ 表示子类型关系，$f(?)$ 表示类型转换，那么：当 $X \leqslant Y$ 时，$f(X) \leqslant f(Y)$ 成立，则称 $f(?)$ 具有协变性；当 $X \leqslant Y$ 时，$f(Y) \leqslant f(X)$ 成立，则称 $f(?)$ 具有逆变性；如果上述两种关系均不成立，则称 $f(?)$ 具有不变性。——译者注

```
T extends Runnable & AutoCloseable
```

接口边界的数量并无限制，但只能有一个类边界。如果采用某个类作为边界，它必须在所有边界中居于首位。

A.5　Java 8 API示例

接下来，我们将讨论 Java 8 引入的一些新方法。

A.5.1　Stream.max方法

在 java.util.stream.Stream 接口中，max 方法的签名如下：

```
Optional<T> max(Comparator<? super T> comparator)
```

注意 Comparator 中使用的下界通配符。通过应用所提供的 Comparator，max 方法将返回流中最大的元素。由于流在为空时可能没有返回值，max 方法的返回类型为 Optional<T>。如果找到最大值，max 方法将其包装在 Optional，否则返回空 Optional。

为简单起见，考虑例 A-13 显示的 Employee POJO。

例 A-13　简单的 Employee POJO

```
public class Employee {
    private int id;
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // 其他方法
}
```

例 A-14 创建了一个员工集合并转换为 Stream，然后通过 max 方法查找具有最大 id 和最大 name（按字母顺序排序[5]）的员工。实现采用匿名内部类来强调 Comparator 可以是 Employee 或 Object 类型。

例 A-14　查找最大的 Employee

```
List<Employee> employees = Arrays.asList(
    new Employee(1, "Seth Curry"),
    new Employee(2, "Kevin Durant"),
    new Employee(3, "Draymond Green"),
    new Employee(4, "Klay Thompson"));

Employee maxId = employees.stream()
    .max(new Comparator<Employee>() {          ❶
        @Override
```

注 5：严格来说是按字典序（lexicographical order）排序，即大写字母位于小写字母之前。

```
            public int compare(Employee e1, Employee e2) {
                return e1.getId() - e2.getId();
            }
        }).orElse(Employee.DEFAULT_EMPLOYEE);

    Employee maxName = employees.stream()
        .max(new Comparator<Object>() {           ❷
            @Override
            public int compare(Object o1, Object o2) {
                return o1.toString().compareTo(o2.toString());
            }
        }).orElse(Employee.DEFAULT_EMPLOYEE);

    System.out.println(maxId);    ❸
    System.out.println(maxName); ❹
```

❶ `Comparator<Employee>` 的匿名内部类实现

❷ `Comparator<Object>` 的匿名内部类实现

❸ Klay Thompson（最大 ID 为 4）

❹ Seth Curry（最大姓名以字母 S 开头）

我们可以利用 `Employee` 类中的方法编写 `Comparator`，不过仅使用 `Object` 类定义的方法（如 `toString`）同样可行。由于 `max` 方法的定义中使用了通配符 `super`（`Comparator<? super T> comparator`)），`Comparator` 既可以是 `Employee`，也可以是 `Object`。

然而，没有人会这样编写代码。符合 Java 8 习惯用法的实现如例 A-15 所示。

例 A-15　查找最大的 `Employee`（Java 8 习惯用法）

```
    import static java.util.Comparator.comparing;
    import static java.util.Comparator.comparingInt;

    // 创建员工列表

    Employee maxId = employees.stream()
        .max(comparingInt(Employee::getId))
        .orElse(Employee.DEFAULT_EMPLOYEE);

    Employee maxName = employees.stream()
        .max(comparing(Object::toString))
        .orElse(Employee.DEFAULT_EMPLOYEE);

    System.out.println(maxId);
    System.out.println(maxName);
```

上述代码显然更为简洁，但它不像匿名内部类那样强调有界通配符。

A.5.2　Stream.map方法

`Stream` 接口还定义了一个名为 `map` 的方法，它传入 `Function`，包括两个参数，均使用通配符：

```
    <R> Stream<R> map(Function<? super T,? extends R> mapper)
```

map 方法对流中的每个元素（T 类型）应用 mapper 函数，将其转换为 R 类型[6]的一个实例。因此，map 方法的返回类型为 Stream<R>。

由于 Stream 被定义为具有类型参数 T 的泛型类（generic class），map 方法不必在签名中再定义变量 T，但需要使用另一个类型参数 R，以便在返回类型之前出现在签名中。如果 Stream 不是泛型类，map 方法将声明两个参数 T 和 R。

java.util.function.Function 接口定义了两个类型参数，第一个（输入参数）是从 Stream **消费**的类型，第二个（输出参数）是函数**产生**的对象类型。通配符意味着在指定参数时，输入参数必须与 Stream 的类型相同或更高，而输出类型可以是返回流类型的任何子类型。



从 PECS 原则的角度来看，Function 接口的定义或许令人困惑，因为类型是反向的。不过只要记住 Function<T,R> 消费 T 并产生 R，就能理解为何 super 后跟 T，而 extends 后跟 R。

map 方法的应用如例 A-16 所示。

例 A-16　将 List<Employee> 映射到 List<String>

```
List<String> names = employees.stream()
    .map(Employee::getName)
    .collect(toList());

List<String> strings = employees.stream()
    .map(Object::toString)
    .collect(toList());
```

可以看到，Function 声明了两个泛型变量，分别用于输入和输出。在第一个代码段中，方法引用 Employee::getName 使用流中的 Employee 作为输入，并返回 String 作为输出。

在第二个代码段中，由于通配符 super 的缘故，程序将输入变量作为 Object（而非 Employee）的方法处理。输出类型原则上可以是包含 String 子类的 List，但由于 String 被声明为 final，不存在任何子类。

接下来，我们讨论 Java 8 引入的部分方法签名。

A.5.3　Comparator.comparing方法

例 A-15 使用了 Comparator 接口定义的静态方法 comparing。Comparator 接口从 Java 1.0 起就已存在，开发人员或许惊讶于该接口目前包含的方法是如此之多。Java 8 将**函数式接口**定义为包含单一抽象方法（single abstract method）的接口。Comparator 属于函数式接口，所包含的单一抽象方法为 compare，它传入两个均为泛型类型 T 的参数。根据第一个参数小于、等于或大于第二个参数，compare 方法将分别返回负整数、0 或正整数[7]。

注 6：Java API 使用 T 表示单个输入变量，或 T 和 U 表示两个输入变量，以此类推。API 通常使用 R 表示返回变量。而对于映射，API 使用 K 表示键，V 表示值。

注 7：有关比较器的讨论请参见范例 4.1。

而 comparing 方法的签名如下：

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(
    Function<? super T,? extends U> keyExtractor)
```

观察 comparing 方法的参数可以看到，其名称为 keyExtractor，类型为 Function。与之前类似，Function 定义了两个泛型类型，分别用于输入和输出。输入的下界由输入类型 T 指定，而输出的上界由输出类型 U 指定。参数名在这里作为键使用：函数采用某种方法提取出需要排序的属性，comparing 方法通过返回 Comparator 来完成这项工作。

我们希望使用给定属性 U 对流排序，因此 U 必须实现 Comparable。换言之，在声明 U 时，U 必须要继承 Comparable。当然，Comparable 本身是一种类型化接口（typed interface），其类型通常为 U，但也可以是 U 的任何超类。

comparing 方法最终返回的是 Comparator<T>，然后 Stream 接口定义的其他方法使用 Comparator<T> 对流排序，结果流与原始流的类型相同。

comparing 方法的用法请参见例 A-15。

A.5.4 Map.Entry.comparingByKey与Map.Entry.comparingByValue方法

最后，我们编写程序将员工添加到 Map（键为员工 ID，值为员工姓名），并根据 ID 或姓名进行排序，然后打印结果。

第一步是将员工添加到 Map。借由静态方法 Collectors.toMap，只需一行代码就能实现：

```
// 使用ID作为键，将员工添加到映射
Map<Integer, Employee> employeeMap = employees.stream()
    .collect(Collectors.toMap(Employee::getId, Function.identity()));
```

toMap 方法的签名如下：

```
static <T, K, U> Collector<T, ?, Map<K, U>> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper)
```

Collectors 是一种工具类（仅包含静态方法），提供 Collector 接口的实现。

从 toMap 方法的签名可以看到，它传入两个函数作为参数，一个用于生成键，另一个用于在输出映射中生成值。toMap 方法的返回类型为 Collector，它定义了三种泛型参数。

根据 Javadoc 的描述，Collector 接口的签名如下：

```
public interface Collector<T,A,R>
```

三种泛型类型的定义如下。

- T：归约操作的输入元素类型；
- A：归约操作的可变累加类型（通常隐藏为实现细节）；
- R：归约操作的结果。

Employee::getId 相当于 toMap 方法签名中的 keyMapper。换言之，T 是 Integer；而结果 R 是 Map 接口的实现，它使用 Integer 替换 K，Employee 替换 U。

有意思的是，Collector 接口定义中的变量 A 是 Map 接口的实际实现。它可能是 HashMap[8]，但我们不得而知，因为结果用作 toMap 方法的参数，无法被观察到。不过在 Collector 中，类型使用无界通配符 ?，这意味着类型在内部要么仅使用 Object 类中的方法，要么使用 Map 接口中不特定于类型的方法。实际上，在调用 keyMapper 和 valueMapper 函数后，类型仅使用 Map 接口新增的默认方法 merge。

为实现排序，Java 8 为 Map.Entry 接口引入了静态方法 comparingByKey 和 comparingByValue。如例 A-17 所示，程序根据键对映射元素排序，然后打印结果。

例 A-17　根据键对映射元素排序并打印结果

```
Map<Integer, Employee> employeeMap = employees.stream()
    .collect(Collectors.toMap(Employee::getId, Function.identity())); ❶

System.out.println("Sorted by key:");
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByKey())
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue()); ❷
    });
```

❶ 使用 ID 作为键，将员工添加到 Map

❷ 根据 ID 对员工排序，然后打印结果

comparingByKey 方法的签名如下：

```
static <K extends Comparable<? super K>,V>
    Comparator<Map.Entry<K,V>> comparingByKey()
```

comparingByKey 方法不传入任何参数，它返回一个比较 Map.Entry 实例的 Comparator。由于我们根据键比较员工姓名，键 K 的声明泛型类型必须是 Comparable 的子类型，才能执行实际的比较操作。当然，Comparable 本身定义了泛型类型 K 或 K 的某种父类型，这意味着 compareTo 方法可以使用 K 类（或更高）的属性。

根据键进行排序的结果如下：

```
Sorted by key:
1: Seth Curry
2: Kevin Durant
3: Draymond Green
4: Klay Thompson
```

根据值进行排序则有些复杂。如果不了解泛型类型的相关知识，就很难理解错误的成因。comparingByValue 方法的签名如下：

```
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>>
    comparingByValue()
```

注 8：在引用实现中的确是 HashMap。

与 comparingByKey 方法不同，在 comparingByValue 方法中，V 必须是 Comparable 的子类型。

根据值排序时，很容易写出下面这样的代码：

```
// 根据员工姓名排序，然后打印结果（无法编译）
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue());
    });
```

不过代码无法编译，程序会提示错误：

```
Java: incompatible types: inference variable V has incompatible bounds
    equality constraints: generics.Employee
    upper bounds: java.lang.Comparable<? super V>
```

原因在于映射中的值是 Employee 的实例，但 Employee 并未实现 Comparable。好在 comparingByValue 方法还包括一种重载形式：

```
static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(
    Comparator<? super V> cmp)
```

comparingByValue 方法传入 Comparator 作为参数，并返回一个新的 Comparator，它根据值比较各个 Map.Entry 元素。对映射值排序的正确方式如例 A-18 所示。

例 A-18　根据值对映射元素排序并打印结果

```
// 根据员工姓名排序，然后打印结果
System.out.println("Sorted by name:");
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByValue(Comparator.comparing(Employee::getName)))
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue());
    });
```

通过为 comparing 方法提供方法引用 Employee::getName，就能实现按员工姓名的自然顺序排序：

```
Sorted by name:
3: Draymond Green
2: Kevin Durant
4: Klay Thompson
1: Seth Curry
```

希望上述示例能提供足够的背景知识，以免读者在阅读和使用 Java API 时对泛型感到困惑。

A.5.5　类型擦除

使用 Java 这样的语言开发时，如何保持长久以来的向后兼容性让人颇费脑筋，开发团队为此做了不少努力。以泛型为例，与泛型有关的信息将在编译阶段被删除，从而不会为参数化类型创建新的类，避免了可能出现的运行时错误。这称为类型擦除（type erasure）。

由于所有操作均在后台完成，开发人员真正需要了解的是在编译时：

- 有界类型参数被替换为参数边界；
- 无界类型参数被替换为 Object；
- 在需要时插入类型强制转换；
- 生成桥接方法（bridge method）以保持多态（polymorphism）。

对类型而言，结果相当简单。Map 接口定义了两种泛型类型，其中 K 代表键，V 代表值。在实例化 Map<Integer,Employee> 时，编译器分别用 Integer 和 Employee 替换 K 和 V。

从 Map.Entry.comparingByKey 方法的签名可以看到，键被声明为 K extends Comparable，这使得类中所有出现的 K 都会被替换为 Comparable。

Function 接口定义了两种泛型类型 T 和 R，所包含的单一抽象方法为：

```
R apply(T t)
```

从 Stream.map 方法的签名可以看到，其边界为 Function<? super T,? extends R>。观察例 A-16 中的 map 方法：

```
List<String> names = employees.stream()
    .map(Employee::getName)
    .collect(Collectors.toList());
```

Function 采用 Employee 替换 T（因为这是一个由员工构成的流），采用 String 替换 R（因为 getName 的返回类型为 String）。

关于类型擦除的讨论大致如此，但某些极端情况并未考虑在内。感兴趣的读者可以参考 Java 官方教程（Java Tutorials），不过类型擦除或许是所有技术中最简单的概念。

A.6 小结

Java 1.5 引入的泛型概念目前依然存在，不过随着 Java 8 的兴起，相应的方法签名变得更为复杂。在 Java 中，大部分函数式接口同时使用泛型类型和有界通配符以强化类型安全。希望读者能通过本附录了解泛型的基础知识，从而在实际开发中正确应用 API。

作者简介

Ken Kousen 是知名的技术培训讲师、软件开发者与会议演讲者，对 Java 以及 Android、Spring、Hibernate/JPA、Groovy、Grails、Gradle 等开源系统颇有研究。除《Java 攻略》外，Kousen 还是 *Gradle Recipes for Android*（O'Reilly Media 出版）与 *Making Java Groovy*（Manning Publications 出版）两本图书的作者，并为 O'Reilly Media 录制了 Android、Groovy、Gradle、Grails 3、Advanced Java、Spring 框架等多门视频课程。

Kousen 活跃于世界各地的技术大会，曾受邀在美国亚特兰大举行的 DevNexus[1] 发表主题演讲，并作为核心开发者在美国明尼阿波利斯、丹麦哥本哈根以及印度新德里举行的 GR8Conf[2] 进行演讲。2013 年和 2016 年，Kousen 荣膺 JavaOne Rock Star 大奖[3]。

Kousen 具有深厚的学术背景，拥有麻省理工学院机械工程与数学学士学位、普林斯顿大学航空航天工程硕士与博士学位，并于伦斯勒理工学院（Rensselaer Polytechnic Institute）[4] 取得计算机科学硕士学位。Kousen 目前担任 Kousen IT 公司总裁，该公司位于美国康涅狄格州。

封面介绍

本书封面上的动物是水鹿（sambar，学名：Rusa unicolor），这是一种原产于南亚的大型鹿种，往往聚集在河流附近。成年水鹿的肩高为 40 到 63 英寸（102 到 160 厘米），体重一般为 200 到 700 磅（91 到 318 千克），雄鹿的体型明显大于雌鹿。水鹿是继麋鹿和驼鹿之后现存的第三大鹿种。

水鹿主要在黄昏或夜间活动，族群的规模通常较小。雄鹿在一年的大部分时间里喜欢独居，而最小的雌鹿群体可能只有三只。较之其他鹿种，水鹿表现出更强的双足能力，可以吃到更高的枝叶并标记领地，也能对捕食者产生威吓。雌性水鹿保护幼崽的能力在所有鹿种中无出其右，它们更喜欢借助水域来保护自己，因为在水中能有效发挥身高优势以及强大的游泳能力。

2008 年，世界自然保护联盟濒危物种红色名录（IUCN Red List）将水鹿列入“易危”物种。雄鹿鹿角因适合作为战利品与传统药材而备受追捧，这导致了对水鹿的过度捕猎；加之工农业发展对其栖息地的侵蚀，使得水鹿数量在全球范围内呈下降趋势。然而，尽管亚洲的水鹿种群在持续减少，但从 19 世纪后期水鹿进入新西兰和澳大利亚以来，其数量却在稳步增长，目前已在一定程度上对当地的濒危植物物种构成了威胁。

O'Reilly Media 图书封面上的不少动物都濒临灭绝，所有动物对我们的世界都至关重要。如果读者希望了解如何为拯救濒危物种贡献自己的力量，请访问 The O'Reilly Animals。

本书封面图片取自 Richard Lydekker 编著的 *The Royal Natural History*。

注 1：全美第二大 Java 技术大会。DevNexus 2018 于 2018 年 2 月 21 日至 23 日在亚特兰大举行，包括近 20 场主题演讲与研讨会，涵盖 Java、JVM 语言、架构、框架、安全、性能、移动开发、云计算、微服务、FaaS、敏捷开发等各个领域。——译者注

注 2：旨在推广 Groovy 的技术大会，为世界各地的 Groovy 开发人员提供相互交流的平台。GR8Conf 目前包括三场会议，分别在美国（GR8Conf US）、丹麦（GR8Conf Europe）与印度（GR8Conf India）举行。——译者注

注 3：由参会人员评选出的“明星演讲者”，旨在表彰他们为社区和 JavaOne 大会所做的贡献。Kousen 以 *Groovy and Java 8: Making Java Better*（2016 年）和 *Making Java Groovy*（2013 年）为题的演讲获得当年的 JavaOne Rock Star 大奖。——译者注

注 4：伦斯勒理工学院简称 RPI，建于 1824 年，是英语国家历史最悠久的理工科大学，美国 25 所“新常春藤盟校”（New Ivies）之一。——译者注



微信连接



回复“Java”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Java攻略：Java常见问题的简单解法

本书以范例形式编写，涵盖Java 8和Java 9的新特性，旨在让读者掌握如何利用这些新特性来解决开发中遇到的各种问题。

- lambda表达式和方法引用的基础知识
- java.util.function包定义的各种接口
- 用于数据转换和筛选的流操作
- 用于流数据排序和转换的比较器与收集器
- 综合运用lambda表达式、方法引用与流解决实际问题
- 在Optional类中创建实例并提取值
- 支持函数式流的I/O功能
- 取代java.util.Date和java.util.Calendar类的Date-Time API
- 处理并发和并行的新机制

肯·寇森（Ken Kousen），独立咨询师与培训讲师，Kousen IT公司总裁；对Spring、Hibernate、Groovy、Grails等语言和框架颇有研究；荣膺2013年和2016年JavaOne Rock Star大奖；毕业于MIT并取得了普林斯顿大学博士学位。

“如果希望了解如何应用Java的新特性解决日常工作中遇到的问题，本书当属明智之选。Ken Kousen对各种常见问题条分缕析，并将解决方案以简练的形式呈现给读者。”

——**Venkat Subramaniam博士**
Agile Developer公司创始人

“本书提供了一种帮助读者快速有效了解Java最新发展的好方法。对所有希望提高自身知识水平的Java开发人员而言，本书都堪称良师益友。”

——**Trisha Gee**
Java Champion、
JetBrains公司Java布道师

PROGRAMMING

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-48880-0
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks